Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080520

HFM: Hybrid File Mapping Algorithm for SSD Space Utilization

Jaechun No[1,*], *Soo-Mi Choi*[1], *Sung-Soon Park*[2] *and Cheol-Su Lim*[3]

[1] College of Electronics and Information Engineering, Sejong University, 98 Gunja-dong, Gwangjin-gu, 143-747, Seoul, Korea
[2] Dept. of Computer Engineering, Anyang University and Gluesys Co. LTD, Anyang 5-dong, Manan-gu, Anyang, 430-714, Korea
[3] Dept. of Computer Engineering, Seokyeong University, 16-1 Jungneung-dong, Sungbuk-gu, 136-704, Seoul, Korea

Received: 23 Aug. 2013, Revised: 20 Nov. 2013, Accepted: 21 Nov. 2013
Published online: 1 Sep. 2014

Abstract: Although the technology of flash memory is rapidly improving, SSD usage is still limited due to the high cost per storage capacity. An alternative is to construct the hybrid structure where a small SSD partition is combined with the large HDD partition, to expand the file system space to HDD storage capacity while exploiting the performance advantage of SSD. In such a hybrid structure, increasing the space utilization of SSD partition is the critical aspect in generating high I/O performance. In this paper, we present HFM (Hybrid File Mapping) that has been implemented for the hybrid file system integrated with SSD. HFM enables to divide SSD partition into several, logical data sections with each composed of the different extent size. To minimize fragmentation overhead, HFM defines three different ways of partitioning functions based on the extent size of each data section. Furthermore, file allocations on the extent are performed on the partitioned unit (segment) of extents to reuse the remaining free space as much as possible. The experimental result of HFM using three public benchmarks shows that HFM can be effective in increasing the usage of SSD partition and enables to contribute to provide better I/O throughput.

Keywords: Hybrid file mapping, partitioning function, map function, map table, allocation unit

1 Introduction

Many file systems have been developed for the purpose of optimizing the moving overhead of HDD disk arms in I/O operations. However, as the new technologies, such as SSD (Solid State Device), are rapidly improved, the file system research for integrating those technologies into the storage capacity has become received the great attention because they possess the promising performance characteristics to satisfy the need of commercial applications. Especially, SSD that uses flash memory as the storage medium is considered the next-generation storage device, due to its advantages such as high random I/O speed and non-volatility[1].

Although SSD reveals the peculiar device characteristics that do not take place in HDD, such as wear-leveling and block erasure [4,13], several researches to overcome those obstacles have been performed either by implementing FTL (Flash Translation Layer)[14,21], or by implementing flash-specific file systems[9,39]. However, the major impediment in utilizing the performance advantage of SSD for constructing the large-scale storage space with only SSD still remains, which is the high ratio of cost per capacity as compared to that of HDD[31,36].

An alternative for solving the problem is to provide the hybrid file system space in which both HDD and SSD partitions are integrated in a single, virtual file system address space in the cost-effective way. In the hybrid structure, the major consideration is to maximize the space utilization of SSD partition while efficiently arranging SSD address space to generate the better I/O bandwidth.

In this paper, we present the hybrid file mapping method, called HFM (Hybrid File Mapping), where the file system space is provided by constructing the hybrid structure with HDD and SSD partitions. To maximize the space utilization of SSD partition, HFM supports the capability of dividing the address space of SSD partition into multiple, logical data sections with each composed of the different I/O unit (extent). The mapping between files and data sections can be performed by considering the file access characteristics including file size or usage.

* Corresponding author e-mail: jano@sejong.ac.kr

Furthermore, such a file mapping can be changed without affecting the directory hierarchy.

In order to reduce the fragmentation overhead in providing the elastic extent size, file allocations in the data section are performed in the units of segments consisting of extents, which enables to reuse extents containing the remaining free space as much as possible. By evaluating HFM with three public benchmarks including TPC-C, PostMark, and IOzone, we tried to show the effectiveness of HFM in generating high I/O performance. This paper is organized as follows: In section 2, several researches related to SSD and file systems are presented. The detailed structure of HFM is described in section 3. In section 4, the experimental results using the benchmarks are presented. Finally, we conclude in section 5.

2 Related Studies

The SSD uses NAND flash memory for the storage medium. Each flash memory is composed of a set of blocks that are the erase units[1,2,22,35] and again a block is consisted of a set of pages that are IO units. The flash memory only allows erase-write-once behavior. Therefore, to update to the same location, a free block is selected and then the new data is written to the block, along with copying the live data in the original block. The original block is erased for the block reclamation. As blocks are involved in the erase operations they become worn out and also the lifetime of a flash block for which it guarantees data reliability is limited (100K for SLC and 10K for MLC)[27]. As a result, evenly distributing the erase-write cycles among flash blocks is the critical issue in SSDs, which is called wear-leveling[13,37].

The wear-leveling is deeply related to the logical to physical address mapping since the block worn-out depends on how often each block is used for writing data. In general, there are two kinds of address mappings to convert the logical addresses to physical addresses: page-level mapping and block-level mapping. Although the address translation speed of the page-level mapping is faster than that of block-level mapping, it suffers from the large memory requirement to store the map table. On the other hand, the block-level mapping suffers from the slow translation speed due to the page-copy operation[29,40].

To overcome the disadvantages of both mappings, several hybrid address translations were proposed. In the log block-based mapping[21], a few number of log blocks are reserved to collect flash pages. When all the log blocks are exhausted, the pages in the log blocks are merged with the corresponding data blocks. However, the log block-based mapping can suffer from the low space utilization due to the limited number of log blocks, resulting in the frequent write and erase operations.

The fully-associative section translation[25] tried to solve the low space utilization by eliminating the concept of dedicating each log block to a specific data block. The log block can be used by pages belonging to any data blocks so that the merge with the data block can be delayed to reduce the write and erase costs. However, if the pages in the log block belong to the different data blocks, then the method can cause the significant erase operations to merge with the data blocks.

Another way of increasing the space utilization of log blocks is to group data blocks into N data groups and also to provide K log blocks at maximum for each data group. The merge operation between pages in the log block and original data blocks takes place in the dedicated data group[29]. Chiang et al. also proposed data clustering algorithm where data blocks are divided into several regions based on the write access frequency[5]. When a data block is updated, it moves to the upper region. On the other hand, if a segment is selected for cleaning, then all the valid blocks in the segment are copied into the lower region. Therefore, the separation between hot data and cold data can be done to reduce the erase operations.

Wu et al. proposed the adaptive flash translation layer where the page-level mapping and the block-level mapping are used according to the recentness[40]. The page-level mapping is used for the most recently used pages. Since the size of the page-level mapping table is limited, some least recently used pages are moved to the block-level mapping by using LRU list. Also, the dual-pool algorithm proposed by Chang and Du manages data blocks based on the erase count and prevents the old block from being involved in the block reclamation[4].

Also, there were several researches related to the write behavior of SSD. For example, Rajimwale et al. found that the write amplification can be reduced by merging and aligning data to stripe sizes[33]. However, since the related information, such as SSD stripe size, is rarely available to file systems, delegating the block management to SSD and exposing the object-oriented interface, such as OSD[11,12], to file system might be effective.

Birrell et al. proposed a way of improving random writes on flash disks, by introducing the volatile data structures[3]. They tested several UFDs (USB Flash Disks) while varying the distance between consequent writes and found that several UFDs deploy the performance degradation with far distance between writes due to read-modify-write to the new flash location. They proposed the volatile data structures and algorithms to mitigate such an overhead in random writes.

Besides implementing the sophisticated method in FTL, several researches tried to find a way of reducing write and erase costs by rearranging data before passing them to FTL. Most of such researches were to provide LRU list to keep the frequently referenced pages in the buffer since those pages will likely to be used soon. Due to the limited buffer space, the least referenced pages will be written to flash pages. There are several approaches for selecting pages to evict them from the buffer. CFLRU[30] divides the LRU list into two regions based on a window size: working region for including the recently used pages

and clean-first region for including clean pages. The victim for the eviction is the page in the clean-first region. The CFLRU/C, CFLRU/E, and DL-CFLRU/E[41] added more flexibility in choosing the victim, by referencing the access frequency and block erase count. For example, in CFLRU/C, if no clean page is available for the eviction, then the dirty page with the lowest access frequency will be selected as a candidate. On the other hand, in CFLRU/E, the dirty page with the lowest block erase count will be chosen for the eviction.

In CFDC[28], the clean-first region of CFLRU is arranged in two queues: one for the clean queue and the other for the priority queue where dirty pages are clustered. The page linked at the tail of the clean queue is first selected as a victim. If no clean page is available, then the first page in the lowest-priority cluster is selected as a victim. LRU-WSR[17] proposed the page replacement referencing an additional flag, called cold flag. To delay flushing dirty pages, their cold flag is checked. If it is marked, then the page is considered as a victim. In case that a dirty page with its cold flag not being marked is referenced, the page is moved to MRU position while marking the cold-flag.

CCF-LRU[26] is another replacement algorithm using the cold detection. It differentiates the clean pages between hot and cold by using the cold-flag. It first searches the cold clean page for the eviction. If no such page is available, then it chooses the cold dirty page as a victim, instead of choosing the hot clean page. In FAB[15], pages in the buffer are clustered based on the erasure block unit. When the buffer is full, FAB chooses a block containing the largest number of pages as a victim, with the expectation of the switch merge in FTL. In case that the number of maximum pages of multiple blocks is the same, a victim is chosen based on LRU order.

BPLRU[20] also selects a block as a victim. It uses the write buffer inside SSD and arranges LRU list in units of blocks. If a logical sector is referenced, then all sectors belonging to the same block are moved to the front of the list. It also chooses the least recent block for the eviction while flushing all sectors in the victim together to flash memory to reduce the log blocks in FTL.

Although the FTL and replacement algorithms mentioned contribute to reduce the write and erase costs in flash memory, most of them need to have accesses to SSD internals, in order to obtain the block erase count or to use the buffer inside SSD. However, such knowledge or accesses cannot be available when using the commercial SSD products. In HFM, we tried to exploit a way of reducing the flash overhead on VFS layer without requiring the knowledge about SSD internals, except for the flash block size. In our method, given the flash block size is known to HFM, the extent size of each data section can be determined to be aligned to flash block boundaries. In case that the extent size is smaller than the flash block size, before passing data to SSD partition, the extents are collected in the in-core map table.

Many researches[6,10,18,23] tried to reduce the cost for write and erase operations by implementing the log-structured I/O scheme on top of file systems. The log-structured file system[34] was proposed to reduce the data positioning overhead on top of HDD. Instead of modifying data in-place, it provides out-of-place updates by appending the logs at the tail. Since such an I/O pattern well fits to flash memory, many flash file systems have adopted the log-structured I/O in their implementations. JFFS, JFFS2, and YAFFS were implemented based on the log-structured mechanism.

JFFS2[39] organizes logs consisted of a linked list of variable-length nodes. Each node contains file metadata, such as file name and inode number, and a range of data in the file. When the file system is mounted, the nodes are scanned to construct the directory hierarchy and the map between file positions and physical flash addresses. In YAFFS[10], fixed-sized chunks are organized to contain file metadata and data. The head chunk with chunk number zero includes file metadata and is used for constructing the directory hierarchy at file system mount. YAFFS uses the tree-structured map to locate the physical addresses associated with file positions.

TFFS[9] was designed for the small embedded systems. In TFFS, each erase unit is consisted of variable-length sectors and the sectors are divided into two parts: one for including descriptors containing the index to the associated data in the unit and the other for real data. It also uses the logical pointer to reduce the overhead of pointer modification due to the unit erasure. One of the interesting flash file systems is FlexFS[24] where the storage space is constructed in the hybrid structure by using MLC and SLC. The new data is first collected in the write buffer and flushed into MLC or SLC log blocks. If no space for writing data is available, then the data migration takes place to create more free spaces, by moving data from the SLC region to the MLC region.

Conquest[38] also supports the hybrid storage space by combining disk to RAM with battery backup. It stores small files and metadata in RAM and stores large files in disk to favor the performance potential of RAM. The hFS[42] is another hybrid file system that provides the file system space by combining the in-place update pattern of FFS with the out-of-place update of LFS (Log-structured file system). Also, TxFlash[32] provides a new device interface to allow the transactional model. TxFlash exports $WriteAtomic(p_1,..,p_n)$ that enables to issue a sequence of writes in the transaction. The consistency is provided by implementing the new cyclic commit protocol.

Finally, DFS[16] provides a support for the virtualized flash storage layer using fusion-io ioDrive. The virtualized flash storage layer integrates the traditional block device driver with FTL, to provide a thin software layer while favoring the fast direct speed of ioDrive controller. Also, the layer is responsible for the direct data path between DFS and the controller, logical block to physical page mapping, wear-leveling, and data

distribution between multiple ioDrives. However, since the layer is tightly coupled with the ioDrive controller, it may not be portable to be used other than ioDrive. Furthermore, the cost for providing DFS address space is much higher than that of HDD.

The difference between flash file systems and HFM is that HFM was designed to provide a large-scale storage capacity by constructing the hybrid file system space. Furthermore, I/Os in HFM take place in-place even though it collects data in units of segments within extents before passing them to SSD. Also, HFM does not use a single I/O unit for the entire SSD storage space. Instead, the different extent size (I/O unit) can be configured for each logical data section, thus storing files to the different data section with the appropriate extent size is possible.

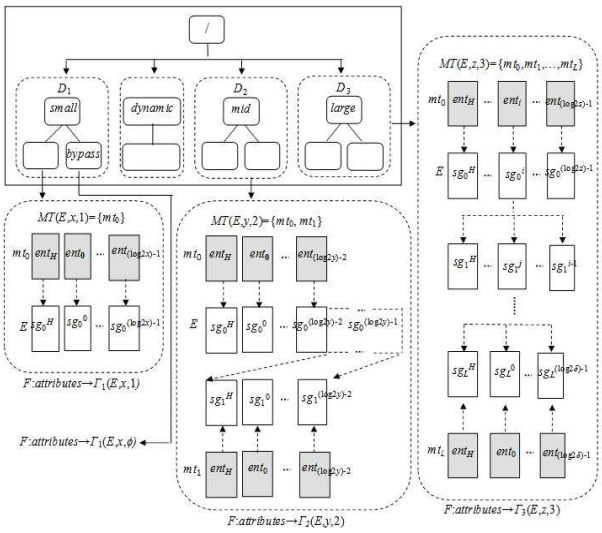


Fig. 1: An overview of HFM

3 System Model

3.1 Overview

The first objective of HFM is to improve the space utilization of the restricted SSD storage resources. Second, HFM has been implemented to reduce I/O cost by mapping files to the appropriate data section while considering their attributes, such as file size, usage, and access pattern. The third objective is to collect as many data as flash block size on VFS layer prior to passing them to SSD partition, in order to reduce FTL overhead in write and erase operations.

In Fig. 1, the entire file system space of SSD partition is composed of three logical data sections. On top of those data sections, their extent size can elastically be defined based on the file attributes. The number of data sections and the extent and section sizes are defined at file system creation. In Fig. 1, three data sections, D_1, D_2, and D_3, are defined with the different extent sizes: x in blocks for D_1, y in blocks for D_2, and z in blocks for D_3 where $x < y < z$.

Definition 1. Given a data section D_n, the associated HFM is defined as $F : attributes \rightarrow \Gamma_n(E,s,\rho)$. The partitioning function Γ_n of D_n is composed of the followings:

- E is the extent of D_n and s is its size in blocks.
- $\rho \in \{1,2,3\} \cup \phi$ is the parameter for the segment partitioning on extents. The $\rho = \phi$ denotes the bypass operation.

In providing with the logical data sections configured with the elastic extent size, the major problem impeding the space utilization is the extent fragmentation overhead. HFM attempts to solve the fragmentation overhead by classifying files according to file attributes and by partitioning the extents to use the remaining space as much as possible. There are two kinds of HFM for a file. In the static HFM, the mapping between a file and data section is defined by the mapping script, which is submitted at file system mount. For instance, in Fig. 1, the files and subdirectories to be created under */small*, */mid*, and */large* are mapped to D_1, D_2, and D_3, respectively, according to the mapping script. In case that file attributes are changed, for instance creating a directory to store large-size files in */small*, the data section being mapped to the new directory can be switched to the other data section consisted of large-size extents, without changing the directory hierarchy.

In the dynamic HFM, the files are mapped in any data section based on file size. Let $len[f]$ be the file size in blocks of a file f. In */dynamic*, if $len[f] \leq x$, then f is allocated in D_1. If $len[f] \geq z$,then f is allocated in D_3. In neither of cases, f is allocated in D_2. Eventually, in HFM, the I/O cost for accessing large-size files can be reduced by assigning large-size extents to them. Also, the files denoting the sequential access pattern, such as multimedia files, can have I/O benefit by using the large-size extents.

On the other hand, the files representing the unpredictable access pattern or size, such as emails, are assigned to small-size extents to minimize the extent fragmentation overhead. Furthermore, the files for backup can bypass SSD partition, resulting in storing only in HDD partition. As a result, the limited SSD space can be used for only files requiring fast I/O bandwidth.

3.2 Segment Partitioning

In this section, we first describe the partitioning function applied to each data section. Since data sections are composed of the different extent sizes, HFM provides the different partitioning function for each data section, to reuse the extents possessing the enough free spaces to allocate several files. Also, we describe the map function

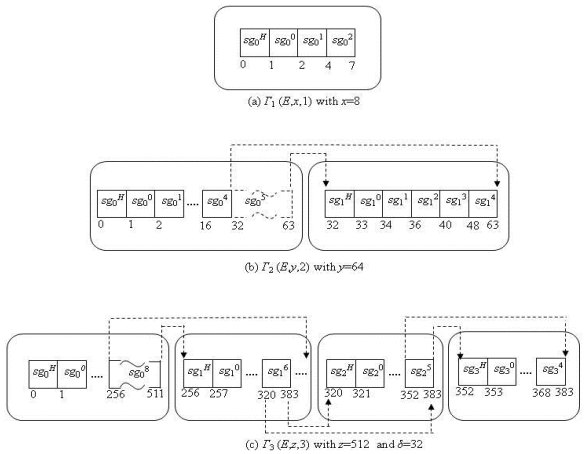


Fig. 2: The partitioning function, according to the extent size and ρ

corresponding to each partitioning function, which is used to determine the block position for the next file allocation.

3.2.1 Partitioning Function

In the partitioning function $\Gamma_n(E,s,\rho)$ with $\rho = 1$ for a data section D_n, E is consisted of $(log_2 s)+1$ segments. In Fig. 1, on top of D_1, $\Gamma_1(E,x,1)$ partitions E into $\{sg_0^i | H \le i < log_2 x\}$ where $H = -1$. The i denotes the segment index in E. If $i = H$, then it is the head segment. Let $start[sg_0^i]$ and $len[sg_0^i]$ be the starting block position and length in blocks of segment i. Then, $start[sg_0^H] = 0$ and $len[sg_0^H] = 1$. Also, $\forall i > H, start[sg_0^i] = 2^i$ and $len[sg_0^i] = 2^i$. The partitioning function for D_1 is defined as

$$\Gamma_1(E,x,1) = \{(sg_0^H,0)\} \cup \{(sg_0^i,2^i) | i \in 0,1,\cdots,(log_2 x)-1\}$$

Example. Fig. 2(a) shows an example of

$$\Gamma_1(E,x,1) \text{ with } x = 8: \{(sg_0^H,0),(sg_0^0,1),(sg_0^1,2),(sg_0^2,4)\}$$

In the partitioning function $\Gamma_n(E,s,\rho)$ with $\rho = 2$, the segment partitioning is performed in two levels. As pictured in Fig. 1, in the partitioning function $\Gamma_2(E,y,2)$ of D_2, E is partitioned in two levels. In level zero, E is consisted of $log_2 y$ segments. To minimize the fragmentation overhead, the largest segment $(log_2 y)-1$ is further split into the child segments in level one, The starting block position and the length in blocks of the segments at level zero are the same as in $\Gamma_n(E,s,\rho)$ with $\rho = 1$. On the other hand, for the child segments of the level one, $start[sg_1^H] = s/2$ and $len[sg_1^H] = 1$. Also, $\forall k > H, start[sg_1^k] = s/2 + 2^k$ and $len[sg_1^k] = 2^k$. Therefore, $\Gamma_2(E,y,2)$ is defined as

$$\Gamma_2(E,y,2) = \{(sg_0^H,0)\} \cup \{(sg_0^i,2^i) | i \in 0,1,\cdots,(log_2 y)-2\} \cup \{(sg_1^H,y/2)\} \cup \{(sg_1^k,y/2+2^k) | k \in 0,1,\cdots,(log_2 y)-2\}$$

Example. Fig. 2(b) shows an example of

$$\Gamma_2(E,y,2) \text{ with } y = 64: \{(sg_0^H,0),(sg_0^0,1),(sg_0^1,2),\cdots,(sg_0^4,16)\} \cup \{(sg_1^H,32),(sg_1^0,33),(sg_1^1,34),\cdots,(sg_1^4,48)\}$$

In the partitioning function $\Gamma_n(E,s,\rho)$ with $\rho = 3$, the segment partitioning is recursively performed into the subsequent level. In Fig. 1, with the partitioning function $\Gamma_3(E,z,3)$ of the data section D_3, as long as the size of segments is no smaller than δ(segment index is $log_2\delta$), the segments are partitioned into the lower level. This partitioning function is used for the large-size extents to consume the free spaces in extents after file allocations.

In the level zero, E is first partitioned into $(log_2 z)+1$ segments whose starting block positions and lengths in blocks are the same as specified in $\Gamma_n(E,s,\rho)$ with $\rho = 1$. The segments whose lengths in blocks are no smaller than δ are recursively partitioned into the lower level until the lengths of all the child segments become smaller than δ.

Let sg_{L-1}^p be a segment at level $L-1$ and $len[sg_{L-1}^p] \ge \delta$. Then, sg_{L-1}^p is partitioned into $\{sg_L^j | H \le j < p\}$. Also, $start[sg_L^H] = start[sg_{L-1}^p]$ and $len[sg_L^H] = 1$. Furthermore, $\forall j > H$, $start[sg_L^j] = start[sg_{L-1}^p] + 2^j$ and $len[sg_L^j] = 2^j$. As a result, $\Gamma_3(E,z,3)$ is defined as

$$\begin{aligned}
&\Gamma_3(E,z,3)\\
&= \{(sg_0^H,0)\} \cup \{(sg_0^i,2^i) | i \in 0,1,\cdots,(log_2 z)-1\}\\
&= \cdots\\
&= \{(sg_L^H, start[sg_{L-1}^p])\} \cup \{(sg_L^j, start[sg_L^H]+2^j) | j \in 0,1,\cdots,p-1\},\\
&\quad \text{if } sg_{L-1}^p \text{ is the parent of those chunks and } len[sg_{L-1}^p] \ge \delta\\
&= \{(sg_{L+1}^H, start[sg_L^j])\} \cup \{(sg_{L+1}^k, start[sg_{L+1}^H]+2^k) | k \in 0,1,\cdots,j-1\},\\
&\quad \text{for } \forall sg_L^j \text{ such that } len[sg_L^j] \ge \delta
\end{aligned}$$

Example. Fig. 2(c) shows an example of $\Gamma_3(E,z,3)$ with $z = 512$ and $\delta = 32$:

$$\{(sg_0^H,0),(sg_0^0,1),\cdots,(sg_0^8,256)\} \xrightarrow{for\ sg_0^8} \{(sg_1^H,256),(sg_1^0,257),\cdots,(sg_1^6,320),\cdots\}$$

$$\xrightarrow{for\ sg_1^6} \{(sg_2^H,320),(sg_2^0,321),\cdots,(sg_2^5,352)\}$$
$$\xrightarrow{for\ sg_2^5} \{(sg_3^H,352),(sg_3^0,353),\cdots,(sg_3^4,368)\}$$

3.2.2 Map Function

Definition 2. Given a HFM $F : attributes \rightarrow \Gamma_n(E,s,\rho)$ of the data section D_n, let pos be the block position on E where the last file has been allocated. If the free space is larger than a threshold value, then $map(pos,\rho)$ determines the starting segment from which the next file is allocated on E.

In $\Gamma_n(E,s,\rho)$ with $\rho = 1$, E is partitioned into $(log_2 s)+1$ segments. Therefore, if the last segment allocated to a file is segment i, then the next allocation starts from segment $i+1$.

case $\rho = 1$:
$map(pos,\rho) = (sg_0^{i+1}, 2^{i+1})$ where $2^i \le pos < 2^{i+1}$

In $\Gamma_n(E,s,\rho)$ with $\rho = 2$, E is partitioned into two levels. In case that an allocation ends a segment $i \le (log_2 s)-2$, the next allocation starts from $i+1$. Otherwise, the last segment $(log_2 s)-1$ is further partitioned into the child segments in level one and the next allocation processes on them.

case $\rho = 2$:
if $2^i \le pos < 2^{i+1}$ and $i \le (log_2 s)-2$,
$map(pos,\rho) = (sg_0^{i+1}, 2^{i+1})$

if $2^i \le pos < 2^{i+1}$ and $i > (log_2 s)-2$,
$map(pos,\rho) = map(lpos = pos - s/2, \rho)$
$= (sg_1^{k+1}, s/2 + 2^{k+1})$
where $\lfloor 2^k \rfloor \le lpos < 2^{k+1}$

In $\Gamma_n(E,s,\rho)$ with $\rho = 3$, E is first split into $(log_2 s)+1$ and any segment whose length in blocks is no smaller than δ (segment index is $log_2\delta$) is further partitioned in the subsequent level. As a result, on the level zero, if the last allocation is completed in segment $i < log_2\delta$, then the next allocation starts from segment $i+1$. Otherwise, segment $i+1$ is further split prior to the next allocation process.

case $\rho = 3$: (level 0)
if $2^i \le pos < 2^{i+1}$ and $i < log_2\delta$,
$map(pos,\rho) = (sg_0^{i+1}, 2^{i+1})$

if $2^i \le pos < 2^{i+1}$ and $i \ge log_2\delta$,
$map(pos,\rho) = map(lpos = pos - start[sg_0^i], \rho)$

(level 1)
if $\lfloor 2^j \rfloor \le lpos < 2^{j+1}$ and $j < log_2\delta$,
$map(lpos,\rho) = (sg_1^{j+1}, start[sg_0^i] + 2^{j+1})$

if $\lfloor 2^j \rfloor \le lpos < 2^{j+1}$ and $j \ge log_2\delta$,
$map(lpos,\rho) = map(lpos = pos - start[sg_1^j], \rho)$

...
(level L)
if $\lfloor 2^k \rfloor \le lpos < 2^{k+1}$ and $k < log_2\delta$,
$map(lpos,\rho) = (sg_L^{k+1}, start[sg_L^H] + 2^{k+1})$

if $\lfloor 2^k \rfloor \le lpos < 2^{k+1}$ and $k \ge log_2\delta$,
$map(lpos,\rho) = map(lpos = pos - start[sg_L^k], \rho)$

Example.

1. In $\Gamma_n(E,s,\rho)$ with $s = 8$ and $\rho = 1$, let $pos = 3$. The next allocation takes place at segment 2:

$$map(3,1) \xrightarrow{2^1 \le 3 < 2^2} (sg_0^2, 4)$$

2. In $\Gamma_n(E,s,\rho)$ with $s = 64$ and $\rho = 2$, let $pos = 35$. The next allocation takes place at segment 2 at level one:

$$map(35,2) \xrightarrow{35-s/2(32)=3} map(3,2) \xrightarrow{2^1 \le 3 < 2^2} (sg_1^2, 32+4=36)$$

3. In $\Gamma_n(E,s,\rho)$ with $s = 512$ and $\rho = 3$, and $\delta = 32$, let $pos = 161$. The next allocation takes place at segment 1 at level two:

$$map(161,3) \xrightarrow{2^7 \le 161 < 2^8, 161 - start[sg_0^7](128)=33} map(33,3) \xrightarrow{2^5 \le 33 < 2^6, 161 - start[sg_1^5](128+32=160)=1} map(1,3) \xrightarrow{2^0 \le 1 < 2^1} (sg_2^1, 160+2=162)$$

Theorem 1. With extents of size s in blocks composed of segments sg_L^i where $i \in \{H, 0, \cdots, (log_2 s)-1\}$, there is at least a sequence of file allocation processes by using HFM of the data section D_n.

Proof. Let u_i be the probability that, after executing the finite partitioning steps, sg_L^i no longer participates in the segment partitioning. Also, let $p^k(\phi)$ be the probability that k partitioning steps take place at sg_L^i and ψ_i be the probability distribution of the segments for n incoming files. Let τ be the used portion of sg_L^i. In case that the remaining space $len[sg_L^i] - \tau$ becomes smaller than δ after k partitioning steps, the segment partitioning at sg_L^i does not take place. Therefore, ψ_i is determined as

$$\psi_i = \begin{cases} \psi_{i-1}u_i, & 0 \le len[sg_L^i] - \tau < \delta \\ p(\phi)\psi_{i-1}, & len[sg_L^i] - \tau \ge \delta \end{cases} \quad (1)$$

For n files,

$$\begin{aligned} \psi_i &= \psi_{i-1}u_i + p(\phi)\psi_{i-1} \\ &= \psi_{i-1}u_i + p(\phi)[\psi_{i-2}u_i + p(\phi)\psi_{i-2}] \\ &= \psi_{i-1}u_i + p(\phi)\psi_{i-2}u_i + p^2(\phi)[\psi_{i-3}u_i + p(\phi)\psi_{i-3}] \\ &= \psi_{i-1}u_i + p(\phi)\psi_{i-2}u_i + p^2(\phi)\psi_{i-3}u_i + p^3(\phi)\psi_{i-3} \\ &= \psi_{i-1}u_i + \cdots + p^k(\phi)\psi_{i-(k+1)}u_i + p^{k+1}(\phi)\psi_{i-(k+1)} \\ &= \psi_{i-1}u_i + \\ &\quad \sum_{r=1}^{k} p^r(\phi)\psi_{i-(r+1)}u_i + p^{k+1}(\phi)\psi_{i-(k+1)} \end{aligned}$$

Because $\psi_{i-1}u_i$ implies no partitioning steps being taken place for n files, it becomes 1. Also, the partitioning steps cannot exceed k, $p^{k+1}(\phi)\psi_{i-(k+1)} \to 0$. Therefore,

$$\psi_i = 1 + \sum_{r=1}^{k} p^r(\phi)\psi_{i-(r+1)}u_i \quad (2)$$

In (2), on average,

$$[\sum_{r=1}^{k} p^r(\phi)\psi_{i-(r+1)}u_i]/n = (\psi_i - 1)/n \quad (3)$$

If n files are all allocated on extents in terms of segments, then $\psi_i \to n$. As a result,

$$[\sum_{r=1}^{k} p^r(\phi)\psi_{i-(r+1)}u_i]/n = (n-1)/n \approx 1 \quad (4)$$

As a consequence, we notice that a file can eventually be allocated in extents by using the segment partition defined in HFM.

3.3 Extent Reuse

In HFM, the extent reuse is performed by using the in-core map table $MT(E,s,\rho)$. The $MT(E,s,\rho)$ is responsible for managing the extent free space in memory for reuse prior to the write operation to SSD partition. Also, given the flash block size of SSD partition is known to HFM, if the extent size of a data section is smaller than the flash block size, then collecting the extents before write operations is performed in the map table.

The $MT(E,s,\rho)$ is constructed per data section and is differently organized according to the extent size and ρ. With $\rho = 1$, $MT(E,s,\rho)$ is consisted of a single table mt_0 to manage the segments split from E. On the other hand, with $\rho = 2$, two tables are included in $MT(E,s,\rho)$: mt_0 for managing the segments in level zero and mt_1 for managing the child segments in level one. Finally, in $MT(E,s,\rho)$ with $\rho = 3$, the segments partitioned at each level are managed by the different table to be organized at the level.

In $MT(E,s,\rho)$, each table entry ent_i is connected to the linked list of extent descriptors whose associated extents have the largest free space starting from segment i. The extent descriptor contains the mapping information about the associated extent, such as extent address, data size mapped to 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 mapped to the extent including inode number.

Fig. 1 shows the $MT(E,s,\rho)$ structure according to the extent size and ρ. Let $E_0(r), E_1(r), \cdots, E_k(r)$ be the largest free spaces of extents $E_0, E_1, \cdots, E_k$ where those free spaces start from segment i at level L. Also, for $\forall k$, let $d(E_k)$ be the extent descriptor of E_k and $start[E_k(r)]$ and $len[E_k(r)]$ be the starting block position and length in blocks of the largest free space in E_k. Suppose that the descriptors are linked together, in the order of $d(E_0), d(E_1), \cdots, d(E_k)$.

Definition 3. Given $MT(E,s,\rho)$, a table entry ent_i of mt_L at level L is defined as:
$ent_i = \{d(E_k) | \forall k \ge 0, start[E_k(r)] = start[sg_L^i]$ *and* $len[E_0(r)] \ge len[E_1(r)] \ge \cdots\}$

In case that the largest free spaces of several extents start from the same segment, their descriptors are linked together in the decreasing order of free spaces. In $\Gamma_n(E,s,\rho)$ with $s = 512$ and $\rho = 3$, suppose that a file of 162 in blocks was allocated in an empty extent. According to the partitioning rule, the next allocation starts from $(sg_2^1, 162)$. The extent descriptor is linked to ent_1 of mt_2 in $MT(E,s,\rho)$. If several extents have the largest free space starting from sg_2^1, then their extent descriptors are linked to ent_1 in the decreasing order of free spaces.

Fig. 3 shows the steps for reusing extents whose free spaces are larger than a threshold θ. The f is the new file to be allocated and *time* denotes the difference between the time for which the associated extent has been inserted into the map table and the time for which the extent is checked for the file allocation. In the algorithm, if the flash block size bs is known to HFM and the extent size s is smaller than bs, then the extent collection takes place in memory before writing to SSD partition. Fig. 4 describes the steps to determine the segment index and level where the next allocation process on the same extent takes place.

HFM maintains the extent bitmap per data section for the extent availability. At file system mount, HFM checks the extent bitmap for each data section and pre-allocates $1K$ of empty extents to the map table. In HFM, file write operations are simultaneously performed on both partitions. When either of partitions completes the write operation, control returns user.

The write completion is denoted by a two bits flag, called *SSD_write_done*, stored in inode: 00 for

Algorithm 1 Extent Reuse
Input: $MT(E,s,\rho)$, f, flash block size bs if available
Output: $MT(E,s,\rho)$

1. HFM periodically organizes $1K$ of empty extents and their descriptors are linked to ent_H to mt_0;
2. **if** $(len[f] \geq s)$
3. use empty extents linked at ent_H in mt_0 for allocating f;
4. let E be the last extent and pos be the last block position allocated to f on E;
5. **else**
6. **for all** $ent_i \in mt_L$ in $MT(E,s,\rho)$
7. select extent E whose largest free space is no smaller than $len[f]$ and $time(E)$ is the longest;
8. let pos be the last block position allocated to f;
9. **end for**
10. **end if**
11. /* id and $level$ are the segment index and level for the next allocation */
12. call $Map(pos,\rho,*id,*level)$;
13. **if** (the remaining space drops below θ)
14. **if** $(s < bs)$
15. collect extents until the total size reaches bs;
16. **end if**;
17. **else**
18. move the extent descriptor of E to ent_{id} of mt_{level} in $MT(E,s,\rho)$;
19. **end if**

Fig. 3: Algorithm for the Extent Reuse

Algorithm 2 Map
Input: pos, ρ
Output: id, $level$

1. **case**
2. $\rho = 1$:
3. calculate i such that $2^i \leq pos < 2^{i+1}$;
4. **return**$(i+1,0)$
5. $\rho = 2$:
6. **if** $(pos \geq s/2)$
7. $lpos = pos - s/2$;
8. calculate k such that $\lfloor 2^k \rfloor \leq lpos < 2^{k+1}$;
9. **return**$(k+1,1)$
10. **else**
11. calculate i such that $2^i \leq pos < 2^{i+1}$;
12. **return**$(i+1,0)$
13. **end if**
14. $\rho = 3$:
15. $level = 0$;
16. calculate i such that $2^i \leq pos < 2^{i+1}$;
17. **if** $(len[sg_0^i] < \delta)$ **return**$(i+1,0)$ **end if;**
18. **while** $(i \geq log_2\delta)$
19. $lpos = pos - start[sg_{level}^i]$; $level++$;
20. calculate i such that $\lfloor 2^i \rfloor \leq lpos < 2^{i+1}$;
21. **if** $(len[sg_{level}^i] < \delta)$ **return**$(i+1,level)$ **end if;**
22. **end while**
23. **end case**

Fig. 4: Algorithm for the map function

initialization, 01 for the completion on HDD partition, 10 for the completion on SSD partition, and 11 for the completion on both partitions. The flag is also used for the file read operation. If the flag for SSD partition is marked as one, then the data is read from SSD partition by using the extent addresses stored in inode. Otherwise, the read operation is performed in HDD partition, followed by the duplication to SSD partition as a background.

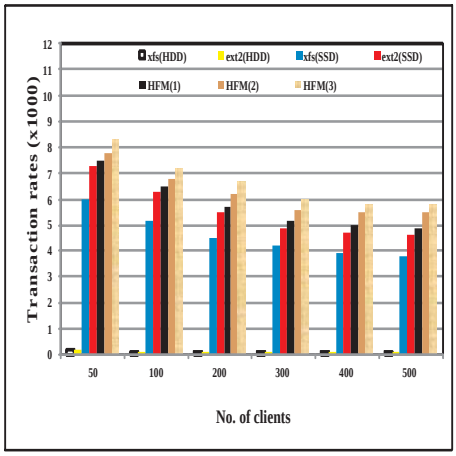
4 Performance Evaluation

4.1 Experimental Platform

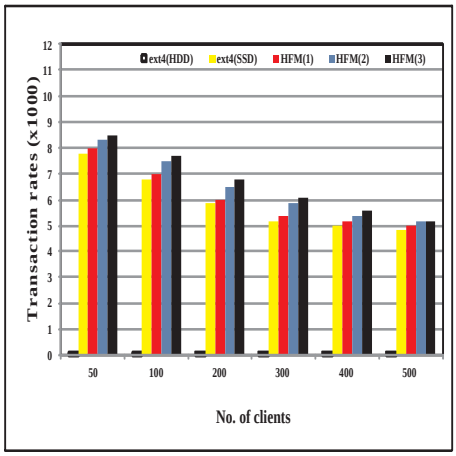
We executed the performance evaluation by using a PC equipped with an AMD Phenom 8650 Triple-core 2.3GHz processor, 4GB of RAM, and Seagate Barracuda 320GB HDD. The SSD is 80GB of fusion-ioDrive [7]. The NAND type of fusion-ioDrive is SLC and the reported bandwidth is 760MB/sec. for reads and 540MB/sec. for writes. The access latency is 26s and it uses PCI-E bus interface. In the evaluation, we divided SSD partition into three logical data sections and mapped to $/hfm/small$, $/hfm/mid$, and $/hfm/large$, which are composed of 8KB, 64KB, and 512KB of extent sizes, respectively. Each data section size is 16GB and HFM block size is set to 1KB. The δ for the segment partitioning is marked as 32. Therefore, with 512KB of extent size, the number of partitioning levels is 5 $(log_2 s/\delta + 1)$, resulting in constructing five map tables. Table 1 represents the HFM and map table associated with each directory hierarchy.

Table 1: HFM definition and map table

HFM(1)	map directory: $/hfm/small$ $F : attributes \rightarrow \Gamma_n(E,s,\rho)$ with $s = 8$ and $\rho = 1$ $MT(E,s,\rho) = \{mt_0\}$
HFM(2)	map directory: $/hfm/mid$ $F : attributes \rightarrow \Gamma_n(E,s,\rho)$ with $s = 64$ and $\rho = 2$ $MT(E,s,\rho) = \{mt_0, mt_1\}$
HFM(3)	map directory: $/hfm/large$ $F : attributes \rightarrow \Gamma_n(E,s,\rho)$ with $s = 512$ and $\rho = 3$ $MT(E,s,\rho) = \{mt_0, mt_1, \cdots, mt_4\}$



(a) HFM integrated with ext2



(b) HFM integrated with ext4

Fig. 5: TPC-C Transaction rates (x1000) with 4GB of RAM

4.2 TPC-C Experiments

TPC-C [8] is the public benchmark for measuring the performance of online transaction processing systems using database. TPC-C typically generates five types of transactions and constructs nine variously structured tables in database. The transactions are randomly generated by a given number of simultaneously connected clients. In TPC-C, we used Mysql 5.5 and the database was installed on both HDD and SSD partitions. We executed TPC-C on top of HFM(1), HFM(2), and HFM(3) and compared their transaction rates. Also, the performance result of HFM is compared to that of ext2, ext4 and xfs installed on HDD and SSD. Each file system uses the database installed on the same partition to minimize the overhead for the database connection.

In Fig. 5(a), the HDD partition of HFM is integrated with ext2 and its transaction rate ($tpmC$) is compared to that of xfs and ext2. Also, in Fig. 5(b), the HDD partition of HFM is integrated with ext4 and its transaction rate is compared to that of ext4. In both figures, the RAM size was configured to 4GB. The number of clients simultaneously connected to database is varied from 50 to 500. As can be seen in the figures, the performance advantage of SSD is apparent because the performance of xfs, ext2 and ext4 installed on SSD shows the large orders of magnitude I/O improvement, compared to that of those three file systems installed on HDD.

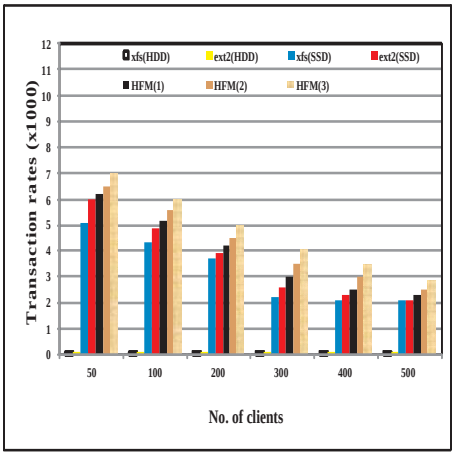
Although the database queries of TPC-C produce various transactions that require to access data from the random disk position, SSD rarely generates the positioning overhead for such I/O patterns. In Fig. 5(a) and 5(b), we can notice that, with 50 client connections, the transaction rate of HFM(3) where the extent size is 512KB is 11% and 6% higher, respectively, than that of HFM(1) where the extent size is 8KB. In the database, the average data size for a single write operation is about 900KB that is larger than any of extent sizes in HFM. Also, since we do not have the information about the flash block size, the steps to collect extents whose sizes are smaller than the flash block size do not take place prior to the write operation. Consequently, the cost for taking the necessary extents in HFM(1) is higher than that in HFM(3).

As the number of client connections increases, the transaction rates become small because of the contention in I/O devices. Even in this case, the performance superiority of SSD is noticeable as compared to that of HDD.

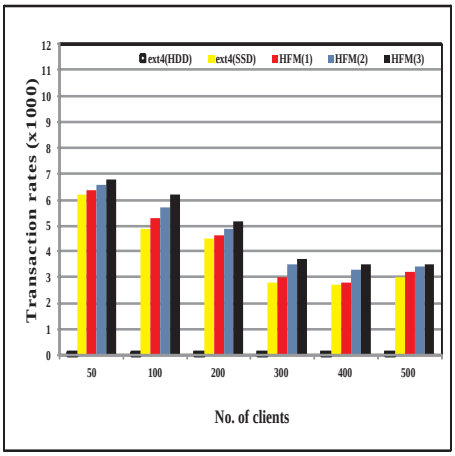
Fig. 6(a) and 6(b) show the transaction rate with 1GB of RAM where the HDD partition of HFM is integrated with ext2 in Fig. 6(a) and with ext4 in Fig. 6(b). Due to the fact that the experiments use the smaller RAM size than that in Fig. 5(a) and 5(b), the virtual I/O activity takes place in processing the transactions. For example, in Fig. 6(a) and 6(b), the transaction rate of HFM(1) with 50 clients decreases about 17% and 20%, respectively, as compared to that with 4GB of RAM in Fig. 5(a) and 5(b). However, the benefit using the large-size extent is still available since, with the same number of connections, HFM(3) integrated with ext2 in Fig. 6(a) shows 12% of speedup, compared to HFM(1). The similar I/O behavior can be observed in Fig. 6(b) where almost 6% of performance improvement can be possible with HFM(3) as compared to HFM(1).

4.3 PostMark Experiments

PostMark has been implemented to measure the performance of short-lived files, such as electronic mail, netnews, and commerce service [19]. As we did with TPC-C, we executed PostMark on three HFM configurations and compared their performance results. The number of files was changed from 1,000 to 20,000 and the file sizes were ranged between 500bytes and

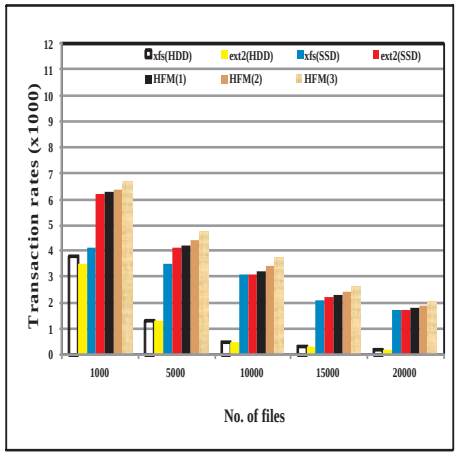


(a) HFM integrated with ext2

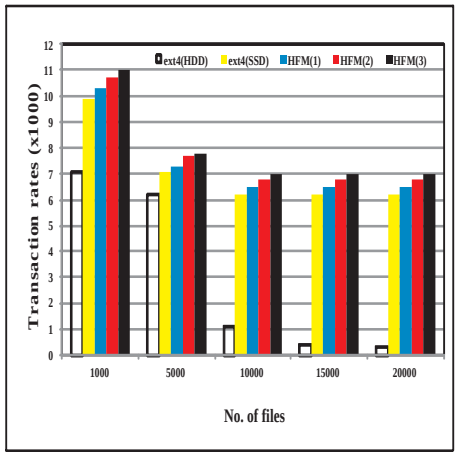


(b) HFM integrated with ext4

Fig. 6: TPC-C Transaction rates (x1000) with 1GB of RAM



(a) HFM integrated with ext2



(b) HFM integrated with ext4

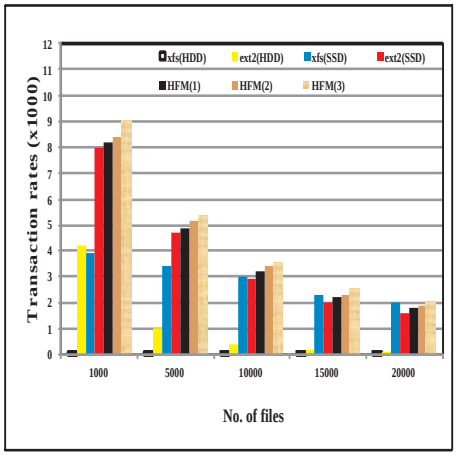
Fig. 7: PostMark Transaction rates (x1000) with 100,000 transactions

9.77Kbytes. In Fig. 7(a), the HDD partition of HFM was integrated with ext2 and in Fig. 7(b), the HDD partition of HFM was integrated with ext4. In both figures, we ran 100,000 transactions and marked *set bias read* as 5, therefore append and read operations are equally likely to occur. As can be seen, with large number of files such as more than or equal to 10,000 files, the performance advantage of SSD is apparent in the file systems due to the rarely generated the mechanical moving overhead in locating the desired data.

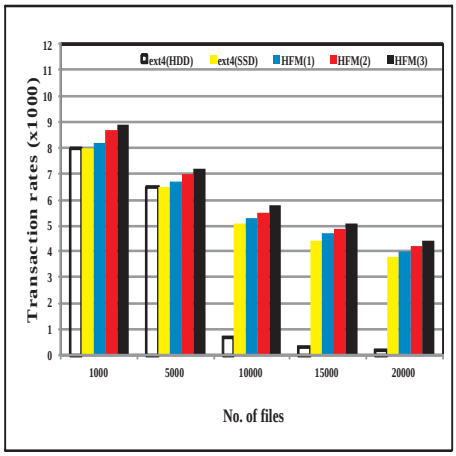
When we executed PostMark on top of HFM(1), it generates the similar bandwidth to ext2 and ext4 installed on SSD. However, on top of HFM(3) with 1000 files, we can observe about 6% performance speedup on ext2 and ext4. As a result, coalescing data into the large I/O granularity on VFS layer would be effective on accessing the continuously being produced small-size files, by reducing the allocation cost. Also, we guess that such an I/O behavior could reduce the FTL bottleneck in SSD partition by converting into the large, sequential I/O access pattern.

In PostMark, the small-size files are continuously generated so that there is little delay in the map table to collect files into the large-size extents. However, if the delay becomes larger, then collecting files in the map table can be an obstacle in achieving high I/O performance. Therefore, choosing the appropriate HFM configuration should carefully be performed to achieve I/O improvement. The transaction rates of the experiments decrease as the number of files becomes large because of the reduced number of the available inodes in the current directory.

Fig. 8(a) and 8(b) show the performance results with 500,000 transactions. In case of xfs, accessing the large number of small-size files does not produce high I/O bandwidth on HDD due to the mechanical moving

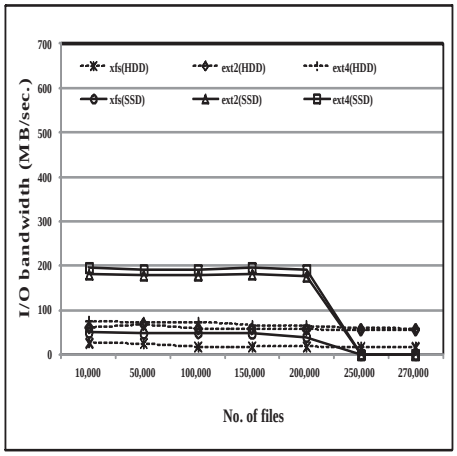


(a) HFM integrated with ext2

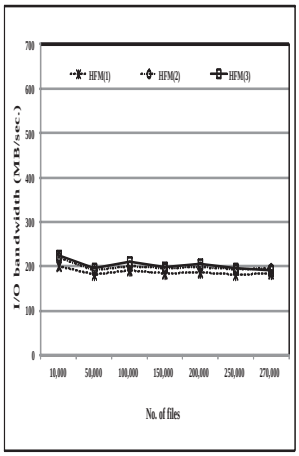


(b) HFM integrated with ext4

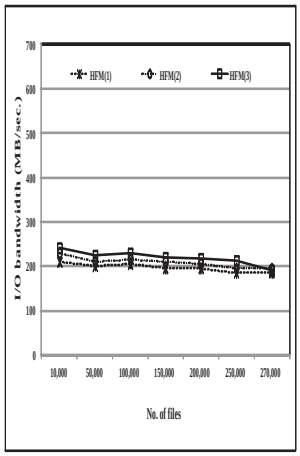
Fig. 8: PostMark Transaction rates (x1000) with 500,000 transactions



(a) ext2, ext4 and xfs



(b) HFM integrated with ext2



(c) HFM integrated with ext4

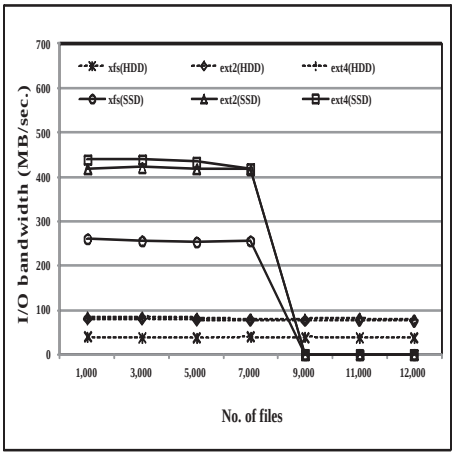
Fig. 9: Type 1 IOzone bandwidth

overhead. However, in Fig. 8(a), we can observe that, with 1000 files, such an overhead can be alleviated in SSD because of its device characteristics. Also, ext2 and HFM all favor such a promising performance benefit on SSD. On the other hand, in Fig. 8(b), with less than 10,000 files, there is little advantage in using SSD on top of ext4 due to its extent metadata structure. However, converting into the large granularity still produces the performance improvement, as can be seen in the bandwidth comparison between ext4 and HFM(3). As the number of files increases the bandwidth decrement takes place due to the large number of files being created in the directory.

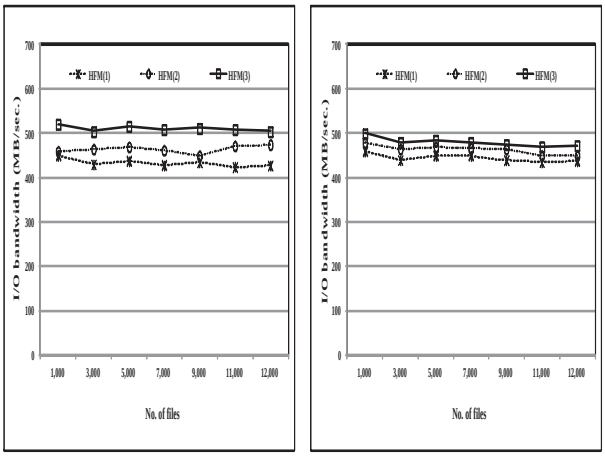
4.4 IOzone Experiments

In this section, we describe the performance results of IOzone benchmark. In the first experiment, we modified IOzone in which about 32GB of HDD space and 16GB of fusion-io SSD space are filled with two types of files. In the first type, 96% (266,240) of small-size files whose sizes are ranged from 8KB to 16KB and 4% (10,200) of large-size files whose sizes are from 512KB to 4MB are randomly generated to occupy both HDD and SSD spaces. In the second type, only the large-size files from 512KB to 4MB are written in two devices. We measured the bandwidth of the two types with HFM(1), HFM(2), and HFM(3). Also, the I/O throughput of ext2, ext4 and xfs installed on HDD and SSD was compared to that of HFM.

The first objective of this experiment is to verify the hybrid structure of HFM by observing whether it can possess the performance advantage of SSD partition while expanding the storage capacity as much as its HDD partition offers. The second objective is to notice the effect of its hybrid file mapping based on the file size.



(a) ext2, ext4 and xfs



(b) HFM integrated with ext2 (c) HFM integrated with ext4

Fig. 10: Type 2 IOzone bandwidth

Fig. 9(a) shows the write throughput of type 1 for ext2, ext4 and xfs installed of HDD and SSD. Also, both Fig. 9(b) and Fig. 9(c) show the result of the same type on HFM integrated with ext2 and ext4, respectively. Although ext2, ext4 and xfs installed on SSD enable to produce high I/O performance, their storage capacity is quickly exhausted as compared to that of the same file systems built on HDD. However, in HFM, since its file system space is integrated with HDD, it can write until the entire HDD partition is occupied with files.

In type 1, the majority is the small-size files and therefore the delay in the map table takes place with the large-size extents, which degrades I/O bandwidth with HFM(3). In Fig. 9(b) and Figure 9(c), we can observe that about 12% and 14% of performance difference occurs between HFM(1) and HFM(3) with 10,000 files, respectively. Also, we can notice that the periodic performance turbulence takes place in HFM due to the extent replacement. We believe that, with the larger SSD capacity, the period between consecutive replacements becomes longer so that the turbulence would be alleviated.

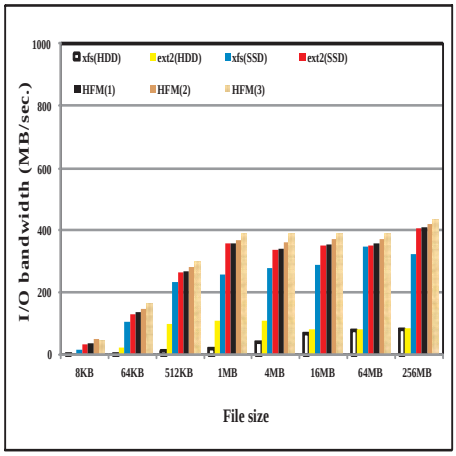
Fig. 10(a) illustrates the write throughput of type 2 for ext2, ext4 and xfs installed on HDD and SSD. In Type 2, the file sizes being written are between 512KB and 4MB. With such file sizes, the storage capacity of SSD is exhausted when the number of files exceeds about 7,500. However, as can be seen in Fig. 10(b) and 10(c), the file system space of HFM is restricted by HDD capacity rather than SSD due to its hybrid structure. In Fig. 10(b), with the large-size files, the performance advantage of HFM(3) is apparent as compared to that of ext2 installed on SSD and that of HFM(1). This is because HFM(3) requires to allocate the less number of I/O units than ext2 and HFM(1) do. For example, with 1,000 files, HFM(3) shows 23% and 16% of performance improvement, compared to ext2 on SSD and HFM(1). We can observe the similar I/O behavior in Fig. 10(c) where about 14% and 9% of bandwidth speedup can be observed with HFM(3) as compared to that of ext4 on SSD and HFM(1).

In the second experiment pictured in Fig. 11(a) and 11(b) where the HDD partition of HFM was integrated with ext2 in Fig. 11(a) and was with ext4 in Fig. 11(b). We varied file sizes and mapped those files to each extent size of HFM, in order to observe what kind of mapping is effective for each file size. In both figures, with 8KB of file size, using HFM(3) composed of 512KB of extent size produces rather less I/O throughput than using HFM(1) composed of 8KB of extent size. This is because 8KB of file sizes should wait at the map table until the whole extent size is used for storing several files. However, with 256MB of files, using HFM(3) shows 6% of improvement in Fig. 11(a) and 3% of improvement in Fig. 11(b) as compared to using HFM(1). This is because it does not cause any delay at the map table and also needs the less number of extents than using 8KB of extents.

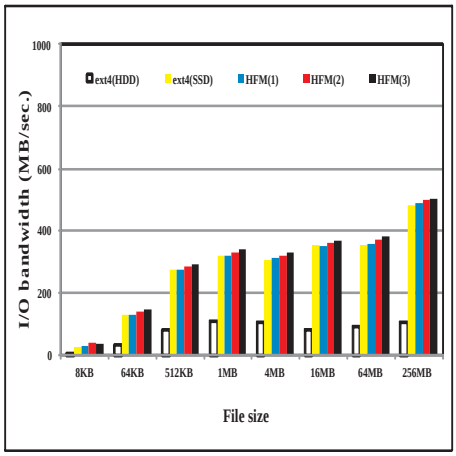
It is noted that the mapping between files and SSD data section should carefully be performed to produce better I/O performance. One efficient way is to use the large-size extent for storing the large-size files. Also, using the large-size extent can be effective even with the small-sized streaming files. However, for the unpredictable-sized files, using the large-size extent may not be efficient due to the delay to fill out the whole extent. In this case, those files should be mapped to SSD data section configured with the small-size or medium-size extent.

5 Conclusion

Due to its promising advantages, integrating SSD into the storage capacity is becoming the main issue in the file system development. Although its peculiar device characteristics, such as erase-write-once and wear-leveling, can be overcome either by implementing



(a) HFM integrated with ext2



(b) HFM integrated with ext4

Fig. 11: IOzone bandwidth compared to ext2, ext4 and xfs installed on HDD and SSD

an efficient FTL algorithm, or by implementing flash-specific file systems, the high ratio of cost per capacity as compared to HDD remains an obstacle in building the large-scale storage subsystems with only SSDs. An alternative is to construct the hybrid structure where a small SSD partition is combined with the large HDD partition, to provide a single virtual address space while favoring the performance advantage of SSD. In such a hybrid structure, maximizing the space usage of SSD partition is the critical issue in generating high I/O performance. HFM was developed to improve the space utilization of the restricted SSD storage resources. In HFM, SSD partition can be organized into several, logical data sections with each composed of the different extent size. In order to reduce the fragmentation overhead, HFM defines three ways of partitioning functions based on the extent size. In the small-size extent $(\rho = 1)$, the extent of size s is composed of $(log_2 s) + 1$ segments and the file allocation steps are performed in units of segments in the extent. In the medium-size extent $(\rho = 2)$, the last segment is more split into level one to reuse the remaining free space of the extent and then in the large-size extent $(\rho = 3)$, the segments whose segment size is larger than or equal to the threshold(δ) are recursively partitioned until the sizes of all segments become smaller than the threshold. We executed the performance evaluation of HFM by using three public benchmarks, including TPC-C, PostMark, and IOzone. In the evaluation, SSD partition is divided into three logical data sections and each data section is configured with 8KB, 64KB, and 512KB of extents, followed by mapping them to the different directory hierarchies. In TPC-C, the performance advantage of SSD is apparent because the performance of HFM and ext2 and ext4 installed on SSD shows the large orders of magnitude I/O improvement, compared to those file systems installed on HDD. Also, since I/O size in TPC-C is larger than any of extent sizes, using the large I/O granularity produces better I/O bandwidth than using the small-size extent. Such an I/O behavior is also available even though the transaction rates become small due to the increased number of simultaneous client connections. In PostMark experiment, we can notice that, in most cases, SSD works better than HDD in accessing small-size files because of the absence of the mechanical moving overhead in SSD. Also, even with small-size files, using the large-size extent contributes to generate the performance speedup because the delay in the map table is reduced due to the contiguously being generated files. In IOzone benchmark, we can observe that HFM enables to expand its storage capacity to its HDD partition due to the hybrid structure. However, there is a delay in writing small-size files with the large-size extent, therefore the mapping between files and SSD data section should carefully be performed. One efficient way is to use the large-size extent for storing the large-size files or the small-sized streaming files. However, for the unpredictable-sized files, using the large-size extent may not be efficient due to the delay to fill out the whole extent. In this case, those files should be mapped to SSD data section configured with the small-size or medium-size extent. As a future work, we will experiment HFM with real applications where a large number of various-sized files are generated, to verify its effectiveness in generating better I/O performance.

Acknowledgement

This work was supported by the Industrial Convergence Strategic Technology Development Program, Grants No. 10045299 and 10047118, funded by the Ministry of Science, ICT and Future Planning, Korea 2013. Also, this work was partially supported by BRL program through the NRF of Korea (2010-0019373).

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse and R. Panigrahy, Design Tradeoffs for SSD Performance, Proc. USENIX Annual Technical Conference, San Diego, CA, 57-90 (2008).

[2] R. Bez, E. Camerlenghi, A. Modelli and A. Visconti, Introduction to Flash Memory, In Proceedings of the IEEE, **91**, 489-502 (2003).

[3] A. Birrell, M. Isard, C. Thacker and T. Wobber, A Design for High-Performance Flash Disks, ACM SIGOPS Operation Systems Review, **41**, 88-93 (2007).

[4] L.P. Chang and C.D. Du, Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers, ACM Transactions on Design Automation of Electronic Systems, **15**, 1-36 (2009).

[5] M.L. Chiang, P. Lee and R.C. Chang, Using Data Clustering to Improve Cleaning Performance for Flash Memory, Software-Practice and Experience, **29**, 267-290 (1999).

[6] H. Dai, M. Neufeld and R. Han, ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes, Proc. SenSys'04, Baltimore, USA, (2004).

[7] Fusion-io, ioDrive User Guide for Linux, (2009).

[8] Fujitsu Technology Solutions, Benchmark Overview TPC-C, White paper, (2003).

[9] E. Gal and S. Toledo, A Transactional Flash File System for Microcontrollers, Proc. USENIX Annual Technical Conference, Anaheim, CA, 89-104 (2005).

[10] E. Gal and S. Toledo, Algorithms and data structures for flash memories, ACM Computing Surveys (CSUR), **37**, 1-30 (2005).

[11] G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg and J. Zelenka, A Cost-Effective, High-Bandwidth Storage Architecture, Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems, 92-103 (1998).

[12] J. Griffin, S. Schlosser, G. Ganger and D. Nagle, Operating System Management of MEMS-based Storage Devices, Proc. 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, (2000).

[13] J.W. Hsieh, T.W. Kuo and L.P. Chang, Efficient Identification of Hot Data for Flash-Memory Storage Systems, ACM Transactions on Storage, **2**, 22-40 (2006).

[14] Intel Corporation, Understanding the flash translation layer(FTL) specification, Technical Report, (1998).

[15] H. Jo, J. Kang, S. Park, J. Kim and J. Lee, FAB: Flash-Aware Buffer Management Policy for Portable Media Players, IEEE Transactions on Consumer Electronics, **52**, 485-493 (2006).

[16] W. Josephson, L. Bongo, K. Li and D. Flynn, DFS: A File System for Virtualized Flash Storage, Proc. 8th USENIX conference on file and storage technologies (FAST'10), San Jose, CA, (2010).

[17] H. Jung, H. Shim, S. Park, S. Kang and J, Cha, LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. IEEE Transactions on Consumer Electronics, **54**, 1215-1223 (2008).

[18] J. Jung, Y. Won, E. Kim, H. Shin and B. Jeon, FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage, ACM Transactions on Storage, **6**, 1-25 (2010).

[19] J. Katcher, PostMark: A New File System Benchmark, Technical report, Network Appliance, Inc., (1997).

[20] H. Kim and S. Ahn, BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage, Proc. 6th USENIX Symposium on File and Storage Technologies, San Jose, CA, 239-252 (2008).

[21] J. Kim, J.M. Kim, S. Noh, S. Min and Y. Cho, A Space-Efficient Flash Translation Layer for CompactFlash Systems, IEEE Transactions on Consumer Electronics, **48**, 366-375 (2002).

[22] J. Kim, H. Lee, S. Choi and K. Bahng, A PRAM and NAND Flash Hybrid Architecture for High-Performance Embedded Storage Subsystems, Proc. EMSOFT'08, Atlanta, GA, (2008).

[23] C. Lee, S. Baek and K. Park, A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices, IEEE Transactions on Computers, **57**, 1002-1008 (2008).

[24] S. Lee, K. Ha, K. Zhang, J. Kim and J. Kim, FlexFS: A Flexible Flash File System for MLC NAND Flash Memory, Proc. USENIX Annual Technical Conference, San Diego, USA, (2009).

[25] S. Lee, D. Park, T. Chung, D. Lee, S. Park and H.Song, A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation, ACM Transactions on Embedded Computing Systems, **6**, 1-27 (2007).

[26] Z. Li, P. Jin, X. Su, K. Cui and L. Yue, CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory, IEEE Transactions on Consumer Electronics, **55**, 1351-1359 (2009).

[27] A. Olson and D. Langlois, Solid State Drives-Data Reliability and Lifetime. White paper, Imation Inc., (2008).

[28] Y. Ou, T. Harder and P. Jin, CFDC: A Flash-aware Replacement Policy for Database Buffer Management. Proc. fifth International Workshop on Data Management on New Hardware, Province, RI, 15-20 (2009).

[29] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho and J. Kim, A Reconfigurable FTL(Flash Translation Layer) Architecture for NAND Flash based Applications, ACM Transactions on Embedded Computing Systems, **7**, 1-23 (2008).

[30] S. Park, D. Jung, J. Kang, J. Kim and J. Lee, CFLRU: A Replacement Algorithm for Flash Memory, 2006 international conference on compilers, architecture and synthesis for embedded systems, Seoul, Korea, (2006).

[31] M. Polte, J. Simsa and G. Gibson, Comparing Performance of Solid State Devices and Mechanical Disks, Proc. 3rd Petascale Data Storage Workshop held in conjunction with Supercomputing'08, Auxtin, TX, (2008).

[32] V. Prabhakaran, T. Rodeheffer and L. Zhou, Transactional Flash, Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, San Diego, CA, 147-160 (2008).

[33] A. Rajimwale, V. Prabhakaran and J. Davis, Block Management in Solid-State Devices, Proc. USENIX Annual Technical Conference, San Diego, CA, (2009).

[34] M. Rosenblum and J. Ousterhout, The Design and Implementation of a Log-Structured File System, ACM Transactions on Computer Systems, **10**, (1992).

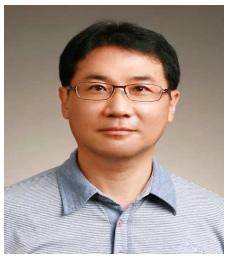
[35] Samsung Electronics, K9XXG08XXM Flash Memory. Technical paper, Samsung Inc., (2007).

[36] M. Saxena and M. Swift, FlashVM: Virtual Memory Management on Flash, Proc. USENIX Annual Technical Conference. Boston, Massachusetts, (2010).
[37] G. Soundararajan, V. Prabhakaran, M. Balakrishnan and T. Wobber, Extending SSD Lifetimes with Disk Based Write Caches, Proc. 8th USENIX Conference on File and Storage Technologies (FAST'10), San Jose, CA, (2010).
[38] A. Wang, G. Kuenning, P. Reiher and G. Popek, The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design, ACM Transactions on Storage, **2**, 1-33 (2006).
[39] D. Woodhouse, JFFS: The Journaling Flash File System. Proc. Ottawa Linux Symposium, Ottawa, (2001).
[40] C. Wu, H. Lin and T. Kuo, An Adaptive Flash Translation Layer for High-Performance Storage Systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, **29**, 953-965 (2010).
[41] Y. Yoo, H. Lee, Y. Ryu and H. Bahn, Page Replacement Algorithms for NAND Flash Memory Storages, Proc. of ICCSA, Kuala Lumpur, Malaysia, 201-212 (2007).
[42] Z. Zhang and K. Ghose, hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance, Proc. EuroSys'07, Lisboa, Portugal, (2007).

Jaechun No received the 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 of the College of Electronics and Information Engineering at Sejong University. Her research areas include file systems, large-scale storage system and cloud computing.

Soo-Mi Choi is currently an Associate Professor in the Department of Computer Engineering at Sejong University, Seoul, Korea. She received her B.S., M.S. and Ph.D. degrees in Computer Science and Engineering from Ewha University of Korea in 1993, 1995 and 2001, respectively. After she received her Ph.D., she joined the Center for Computer Graphics and Virtual Reality (CCGVR) at Ewha University as a Research Professor. In 2002, she became a faculty member in the Department of Computer Engineering at Sejong University. From September 2008 to August 2009 she was a Visiting Scholar at the CG Lab of ETH Zurich in Switzerland. Her current research interests include computer graphics, virtual reality, human-computer interaction, and ubiquitous computing.

Sung-Soon Park received the 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 of 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 the Master's degree from Indiana Univ. and Ph.D. degree in Computer Engineering from Sogang Univ. respectively. 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 of the Dept. Computer Engineering at Seokyeong University. His research areas include multimedia system and cloud computing.