International Journal of Innovative
Computing, Information and Control
Volume 9, Number 2, February 2013

 ISSN 1349-4198
pp. 477–502

AN APPROACH TO EXPLOITING HYBRID FILE SYSTEM SPACE USING NAND-BASED SSD STORAGE RESOURCES

Jaechun No and Yong-Guk Kim

College of Electronics and Information Engineering
Sejong University
98 Gunja-Dong, Gwangjin-Gu, Seoul 143-747, Republic of Korea
{ jano; ykim }@sejong.ac.kr

Received November 2011; revised April 2012

Abstract. *SSD (Solid-State Disk) is a storage device that is considered as an HDD replacement, due to its potentials such as high random I/O performance, reliability and low-power consumption. SSDs are widely being used these days, from small-size mobile equipment to high-end storage subsystems. In this paper, we introduce NF-hybrid (New-Form of hybrid) file system whose primary objective is to utilize performance potentials of SSDs, while addressing their drawbacks. The main obstacle in building large-scale storage subsystems solely composed of SSDs is the higher cost per capacity over HDDs. Furthermore, in applications generating sequential workloads, using disk arrays can deploy a similar performance bandwidth to SSDs. Therefore, blindly adopting SSDs to storage subsystems is not a good way to provide better I/O performance. NF-hybrid exploits a cost-effective way of utilizing SSD's performance advantages while offering a large-scale storage capacity. NF-hybrid's address space is provided with a hybrid structure where both HDD and SSD are integrated through the flexible internal structure. In addition to the hybrid internal structure, NF-hybrid supports the reconfigurable SSD address space by allowing multiple, logical data sections that are composed of the different extent size each. On top of those data sections, files can transparently be mapped to the appropriate data section, by considering their access characteristics. The performance evaluation verifies the effectiveness and suitability of NF-hybrid.*

Keywords: NF-hybrid, Small-size extent, Large-size extent, Extent bitmap, File mapping

1. **Introduction.** As the advantages of SSD have been recognized, such as high I/O performance, reliability and low-power consumption, adopting SSD to IT products is rapidly increasing, from mobile electronics to high-end storage subsystems. The most attractive feature of SSD is that it does not generate mechanical overhead to locate desire data, due to its flash memory components [1,2]. Such promising storage characteristics become the driving force of numerous researches related to SSD, with the expectation of achieving high I/O performance in various environments. For example, database or web applications can obtain performance benefits by employing SSD storage subsystems, to serve a large number of end-user's I/O requests [16].

However, there are several critical constraints in building a large-scale storage subsystem solely composed of SSDs. One of such constraints is that, because of SSD's internal flash memory components, developing a file system for SSD storage subsystems evokes some issues that have not appeared in HDD storage subsystems [3].

First, the flash memory must be erased before data is written to memory [4-6,15]. Since each flash cell can retain valid data only for a limited time period and the cell erasure can quickly wear out memory, the erase/program cycle should carefully be distributed among memory cells, which is called the wear-leveling [7-10]. Furthermore, because I/O

unit (page) differs from erase unit (block) and flash memory does not allow overwrites, file rewrite operations on flash memory might cause the data-copy overhead, to duplicate the valid data stored in the originally mapped block to a free block, along with new data [11-13]. In SSD, those flash-specific operations are performed by FTL (Flash Translation Layer) [12,14].

Another issue is SSD's high cost per capacity, compared with that of HDD. Table 1 shows several commercial SSD products available in the market.

TABLE 1. Commercial SSD products [17]

Drive	Size	Interface	Read B/W	Write B/W	Flash type	Price
Imation SSD (M-class)	64GB	SATAII	150MB/sec.	90MB/sec.	MLC	$220
Imation SSD(S-class)	64GB	SATAII	130MB/sec.	120MB/sec.	SLC	$1100
Intel SSD (X25-M)	160GB	SATAII	250MB/sec.	100MB/sec.	MLC	$425
Intel SSD (X25-E)	64GB	SATAII	250MB/sec.	170MB/sec.	SLC	$730
OCZ (Agility2)	360GB	SATAII	285MB/sec.	275MB/sec.	MLC	$950
OCZ (Vertex2 EX)	200GB	SATAII	285MB/sec.	275MB/sec.	SLC	$1900
Fusion-IO ioDrive	80GB	PCI-e	760MB/sec.	540MB/sec.	SLC	$2400

For applications that deploy sequential workloads, blindly adopting SSDs is not cost-effective because a similar performance bandwidth can be obtained by using disk arrays [18]. An alternative is to build a hybrid storage subsystem where both HDD and SSD are incorporated in an economic manner, while utilizing the strengths of both devices to the maximum extent possible.

In this paper, we introduce NF-hybrid (New-Form of hybrid) file system, which has been developed for hybrid storage subsystems. NF-hybrid was implemented based on two key ideas. First, providing a large-scale space capacity with only SSD devices costs high expenses. Second, the flash memory components of SSD deploy a peculiar semiconductor overhead that has not occurred in HDD storage platform. NF-hybrid proposes a way of exploiting the performance potentials of both SSD and HDD, while offering a large-scale storage capacity in a cost-effective way. The performance evaluation shows that NF-hybrid is capable of generating the comparable I/O performance to file systems installed on SSDs.

This paper is organized as follows. In the next section, we discuss background and related works of SSD. Section 3 describes the detailed description of NF-hybrid. Section 4 presents the performance measurements of NF-hybrid. We conclude in Section 5.

2. Background Works.

2.1. SSD structure.

NAND-based SSD [11,19], as shown in Figure 1(a), consists of host interface, such as SATA or SCSI, SRAM containing tables for address mapping, SDRAM being used for data transmission, flash controller and NAND flash arrays. We used a 80GB of Fusion-io SSD ioDrive for our study, pictured in Figure 1(b) [20].

The storage component of SSD is flash memory. NAND flash memory is divided into two types. SLC (Single-Level Cell) stores a single bit per memory cell, thus deploying only two states: erased state and programmed state. MLC (Multi-Level Cell) allows two bits to be stored in a memory cell; therefore, more than two states are possible to be deployed. Whereas MLC is slower than SLC to program, its capacity is larger than SLC. Also, MLC shows the shorter life time over SLC (10K cycles for MLC and 100K cycles for SLC) and generates more error rates per bit read than SLC ($10^{-5} \sim 10^{-7}$ errors for MLC and $10^{-9} \sim 10^{-11}$ errors for SLC) [11].

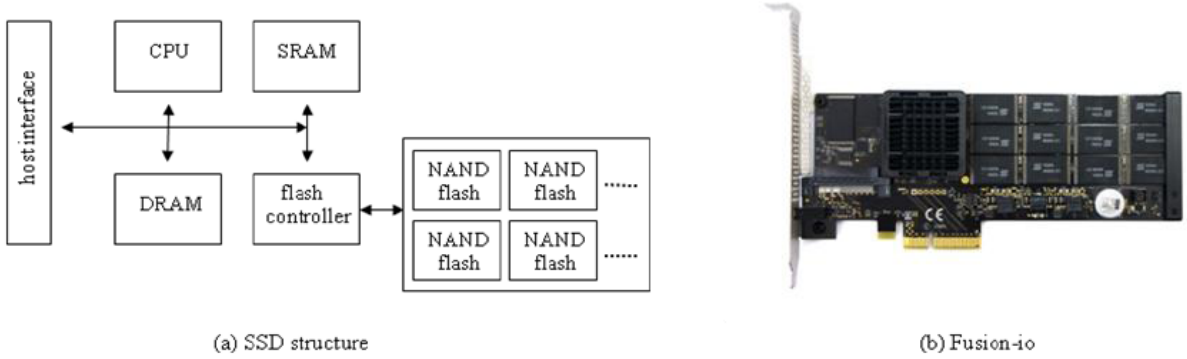

FIGURE 1. SSD structure and Fusion-io

One example of NAND flash memory is Samsung K9KAG08U0M flash chips [19]. Each of those flash chips is organized into four planes, with each plane being divided into two groups, plane $0 \sim 1$ and plane $2 \sim 3$. In each group, even and odd numbered blocks are contiguously distributed. Thus, two-plane page interleaving operations are possible, by dividing the memory cells into plane $0 \sim 1$ and plane $2 \sim 3$. Each plane contains 2048 256KB blocks, in addition to 4224B of page register. The reported block erase time is 1.5ms per block. The block is in turn organized into 64 4KB of pages. The page programming time is 200 μs per page and each page can be read out at 25 ns per Byte.

Even though programming flash memory cell is performed 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 is necessary, to evenly distribute erase/program cycles over memory cells. The wear-leveling process is performed by FTL that emulates a block device driver. The FTL throughput significantly affects the overall system performance. Several wear-leveling algorithms have been proposed by researchers.

For instance, there is a wear-leveling algorithm using log blocks [12,13] in which small writes to blocks are collected in log blocks as long as free pages are available. When the corresponding log block is full, the pages in the log block are merged with the data block and written to flash memory. The wear-leveling algorithm proposed by Chang and Du [21] 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 with 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.

2.2. **Related studies and objectives.** Many studies have been performed to reorder data based on LRU policy before writing to flash memory, in order to reduce the number of write and erase operations. CFLRU [34] tried to minimize the replacement cost by keeping dirty pages in the buffer. It divides LRU list into two parts: working region that contains recently used pages and clean-first region that consists of candidates for eviction. The size of clean-first region is predetermined by a window size w.

If a room for an incoming page is needed, then a victim to be evicted to flash memory is selected from clean pages in the clean-first region. If there is no clean page to be evicted, then the dirty page in the same region is chosen based on LRU order. The page having been re-referenced is moved to the front of the working region. The difficulty of CFLRU is to determine the appropriate window size for various applications. Also, it does not consider the access frequency of victim pages that may be re-referenced in near

future. Finally, it needs the access to flash memory to acquire physical page numbers to be evicted.

LRU-WSR [35] attempts to increase the buffer hit ratio of CFLRU, by considering the page access frequency. LRU-WSR not only postpones writing dirty pages but also checks the access frequency, using the cold detection algorithm that is similar to the second-chance page replacement. When LRU-WSR selects a victim, it checks the cold-flag as well as clean/dirty flag. In case that the dirty page whose cold flag is not set is chosen as a candidate, then the page is moved to the front and selecting a victim continues to the other page in the buffer. If the candidate is a clean page, then it is evicted from the buffer regardless of its status of cold flag. LRU-WSR increases the buffer hit ratio compared with CFLRU by delaying evicting dirty and hot pages. However, since LRU-WSR does not apply the cold-detection step to clean pages, its I/O performance could be decreased if a large number of hot and clean pages are used in applications. Like CFLRU, it also requires to have the access to flash memory for acquiring physical page numbers.

FAB [36] also selects a victim to be flushed, according to LRU policy. However, instead of evicting a page, it chooses a block as a candidate that has the largest number of pages in the buffer. The reason for selecting such a block is to increase the buffer hit for short writes and to increase the chance for switch merge operations. If several blocks have the same largest number of pages, then the victim block is chosen based on LRU policy among them. FAB effectively worked for sequential I/O requests on portable media players. However, with random I/O requests, the performance could be degraded due to lower buffer hit ratio.

BPLRU [8] is similar to FAB in a sense that it also employs a block-level LRU policy. BPLRU maintains the LRU list inside SSD for only write requests. The read requests are simply redirected to FTL. The pages in the write buffer are grouped in units of blocks. When a block is chosen as a victim, all pages belonging to the same block are also flushed to SSD, which can reduce the write and erase costs in log-structured FTL. If a page is re-referenced, the entire pages belonging to the same block are moved to the front due to recency. Like the other schemes mentioned, BPLRU requires to access SSD resources, such as write buffer to retain LRU list, and needs to have knowledge about physical block and page numbers.

FAST [37] tried to reduce the write and erase costs in FTL layer, by solving the disadvantage of log-structured FTL. In log-structured FTL, each write to a data block should be redirected to one log block. Therefore, with the limited number of log blocks, it could suffer from low space utilization that causes frequent writes to the data block in flash memory. FAST attempts to overcome such a disadvantage by spreading write requests to a data block on multiple log blocks.

In FAST, log blocks are divided into two groups: one for sequential writes and the other for random writes. When pages are sequentially written, they are mapped to the sequential log blocks and merged with their original data blocks to erase them. The pages involved in random writes are mapped to any of random log blocks, which could delay merge operation much longer. However, if the pages mapped to a log block are originated from the different data blocks, it can cause the significant number of erase operations to merge with their data blocks.

The aforementioned schemes require the knowledge about the attached flash memory to obtain the physical block and page numbers. However, such knowledge cannot be available unless the internal structure of flash memory is exposed. Unfortunately, many commercial SSD products do not disclose their internal structure to users. In this case, it is not possible to employ the schemes mentioned to reduce write and erase costs of flash memory.

The objective of NF-hybrid is to exploit a way of optimizing FTL overhead in VFS layer, which does not require the access to SSD internal structure. The proposed approach is to collect data in VFS layer prior to write operations, to align the data size with flash block boundaries given that the related information, such as flash block size, is known to NF-hybrid. The data collection of NF-hybrid is integrated with the extent structure that has been designed to reduce fragmentation overhead to maximize SSD storage utilization.

On the other hand, several file system works have provided with the hybrid structure. Conquest [22] tried to minimize disk accesses by using persistent RAM, because the cost for persistent RAM is becoming lower. 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 NF-hybrid come from the distinct storage component integrated with HDD. Unlike SSD, persistent RAM allows in-place data updates and does not deploy semiconductor overhead. As a result, there is little pressure for the data alignment in Conquest. On the other hand, NF-hybrid attempts to align data size with flash block boundaries, to minimize FTL overhead in VFS layer.

Also, Conquest does not support multiple, logical data sections and thus all file allocations should be performed in a single unit size. On the other hand, by considering file access characteristics, NF-hybrid is capable of mapping files to the appropriate data section composed of the different extent size each.

Another example is hFS [23], which combines the advantages of LFS (Log-structured File System) and FFS (Fast File System). LFS [29] supports update-out-of-place in which file updates take place without seeking back to their original location. Although its update behavior is appropriate for flash memory because flash memory does not allow in-place update, the sequential log structure can produce significant I/O overhead in random environments. Furthermore, it incurs a large memory requirement to store in-memory data structure to trace valid blocks [24]. NF-hybrid does not use out-of-place file updates in SSD partition. All file allocations are executed on extent-based, in-place I/O behavior. Also, like Conquest, hFS does not support file mapping considering file access characteristics.

Many flash file systems have tried to overcome flash-related shortcomings, by introducing the concept of log-structured file system [25,28]. For instance, in JFFS and JFFS2 [25], data and metadata are contained in variable-length nodes that are sequentially written in logs. Each node belonging to the same inode contains a version number and the node with the highest version number is considered 'valid'. At file system mount time, the whole flash medium is scanned to build the directory hierarchy by locating valid nodes, which is proportional to the size of flash medium.

Another example is YAFFS [27,32] in which each file is divided into fixed-size chunks and each chunk is marked with file id and chunk number. The chunk with id number zero is a header containing file name and permissions, and the chunks with nonzero id numbers contain file data. File rewrite operations are executed by replacing the relevant chunk to the one with new data. As in JFFS2, file mount requires to scan the entire flash memory.

TFFS [26] is another example of using logs, which was implemented for small embedded systems with less than 4KB of RAM. TFFS targets for NOR devices and provides several useful functions, such as tailored API for the embedded devices and concurrent transaction recoveries. In TFFS, the log is created per erase unit. Each unit contains descriptors identifying their associated data on one side and data on the other side. The mapping between logical-to-physical erase unit is indirectly performed by using logical pointers.

However, many of flash file systems have been developed for small-size flash memory and therefore are not appropriate for large-scale data storage resources. Our main goal in

implementing NF-hybrid is to provide a large-scale storage capacity, while utilizing SSD's performance benefits. The primary objectives in developing NF-hybrid are as follows:

Exploit a flexible, logical disk layout for optimizing SSD address space: In NF-hybrid, SSD partition can be divided into multiple, logical data sections, which are composed of the different extent size each. The file mapping to logical data sections is performed according to file access characteristics and usage. For example, the data section composed of large-size extents can be mapped to large-size files deploying the sequential access pattern, to reduce file access cost. Also, files that do not need fast, interactive I/O service, such as snapshot images, can bypass SSD partition during write operations. In this way, NF-hybrid can effectively manage SSD's tight storage resources, by storing only those files that require high interactive I/O response rate.

Maximize SSD space utilization by using extent partitioning: NF-hybrid provides the extent partitioning to increase SSD space usage. Since the extent can be defined as any size, minimizing extent fragmentation would be critical in increasing SSD usage. To alleviate the fragmentation problem, NF-hybrid partitions extents in units of segments and allocates data according to segments. With extents whose size is larger than a threshold, the largest segment can be further divided into sub-segments, to reduce fragmentation overhead. By combining with in-memory extent bitmap operation using extent table, the partitioning scheme can contribute to increase SSD space utilization, without incurring the significant overhead in I/O operations.

Optimize FTL overhead in VFS layer by adopting data alignment: As mentioned, most algorithms designed to reduce FTL overhead need to obtain some knowledge about the internal structure of flash memory, such as physical block and page numbers. NF-hybrid tries to overcome such a requirement because many commercial SSD products do not disclose their internal structure. In NF-hybrid, given the related information, such as flash block size, is passed to NF-hybrid by users, the extent can be aligned with flash block boundaries in VFS layer. Even though we could not directly measure the effect of data alignment due to the inaccessibility to the embedded FTL modules, we believe that such an alignment can contribute in achieving high I/O performance of NF-hybrid combined with a small portion of SSD partition.

3. Implementation Detail.

3.1. Overall structure.

NF-hybrid is constructed with two different partitions, SSD partition and HDD partition. SSD partition stores hot files recognized by file access time and file system metadata necessary for SSD file allocation, such as extent bitmaps and duplicated inode. On the contrary, HDD partition is used as a backup-ed data storage. To maximize the small portion of SSD storage capacity, NF-hybrid divides SSD logical address space into multiple data sections. The information about those data sections, such as the number of data sections, section and extent sizes, flash block size if available, and the initial directory hierarchy where the logical data section is mapped, is passed to VFS layer by users at file system creation and is stored in in-memory map table.

NF-hybrid uses the different file allocation scheme for two partitions: extent-based allocation for SSD partition and block-based allocation for HDD partition. In the current implementation, HDD partition of NF-hybrid uses the similar file allocation method to ext2, by adopting I/O kernel modules of ext2.

On SSD partition, NF-hybrid allocates new files in the unit of extent. The extent size of each logical data section can differently be defined at file system creation. Such a configuration helps to classify files according to file access characteristics. In NF-hybrid, files can selectively be mapped to the data section composed of the appropriate extent

size, by considering file access pattern, size and usage. For example, the files deploying a large, sequential access pattern can be allocated to the data section with large-size extents, to reduce file allocation cost. On the other hand, the files involved in frequent modifications can be stored in the data section comprised of small-size extents. Also, the files that do not need the immediate interactive I/O service, such as snapshot images, can bypass SSD partition.

Figure 2 illustrates an overview of NF-hybrid. The beginning of SSD partition contains the configuration parameters, such as the number of data sections, section size and extent size of each data section. In Figure 2, SSD partition is divided into three data sections, D_0, D_1 and D_2 in which the extent sizes of those data sections are composed of δ, s and t blocks, respectively, where $\delta < s < t$.

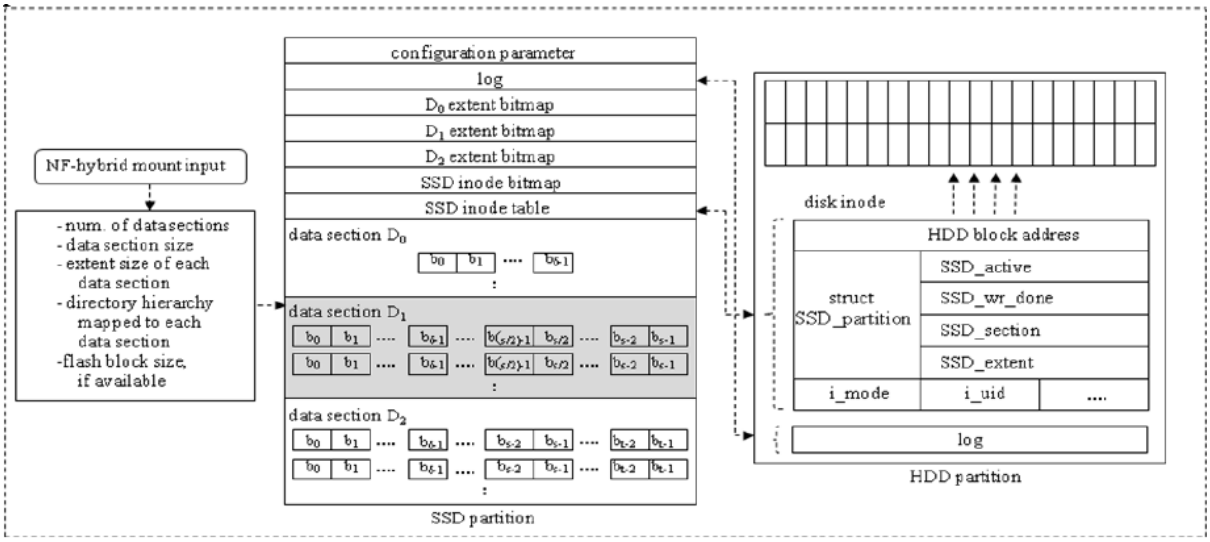

FIGURE 2. NF-hybrid structure

The configuration parameters are immediately followed by logs that are used for synchronizing write operations between two partitions. NF-hybrid also stores logs to HDD partition for the purpose of backup. In case of SSD crash, the logs stored in HDD partition are read to recover I/O transactions on SSD partition. After logs, there exist multiple extent bitmaps. The extent bitmaps are defined per data section and brought into memory at file system mount. Finally, to execute file accesses independently of HDD, NF-hybrid stores the duplicated inode table in SSD partition, together with inode bitmap to allocate inode blocks in SSD partition.

NF-hybrid provides several SSD-related attributes in inode, to integrate both partitions into a single, virtual address space. *SSD_active* and *SSD_wr_done* denote I/O status of the corresponding file. The two bits of *SSD_wr_done* describe four states of file 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.

When *SSD_active* is marked as one, it implies that the associated file is available in SSD partition. This flag is turned off when the file is evicted from the partition due to the extent replacement. *SSD_section* shows the identification of data section where the associated file is mapped and *SSD_extent* includes an array of SSD extent addresses, starting block number in the extent and block count.

3.2. **Extent partitioning.** In NF-hybrid, a single extent is composed of multiple segments. Since NF-hybrid uses the pre-determined extent size, the extent fragmentation problem can limit the available storage capacity of SSD partition. NF-hybrid attempts to alleviate the fragmentation overhead by maintaining bitmap per segment or per sub-segment for an extent. Hereafter, let s be the size of extents in blocks where $s = 2^n (n \geq \log_2 \delta)$.

3.2.1. *Small-size extent.* In case that s is equal to a threshold δ, an extent (small-size extent) is divided into $(\log_2 s) + 1$ segments.

Definition 3.1. *Given a small-size extent E, the structure of E is defined as follows:*

$$E = \{seg[i] | c \leq i < \log_2 s \text{ where } c = -1\}$$

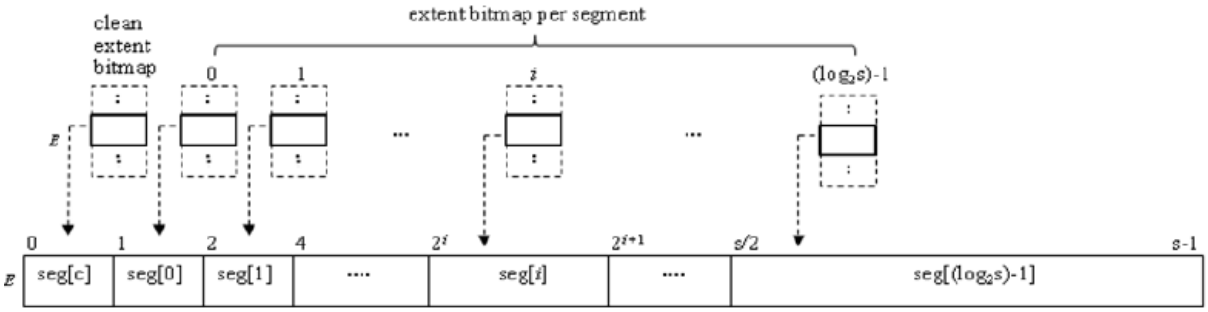

FIGURE 3. Small-size extent structure and bitmaps

Figure 3 shows an overview of small-size extent E. The first segment, $seg[c]$, is the clean segment. The starting block position and the size of segment i, $pos(seg[i])$ and $size(seg[i])$, are both 2^i, except for the clean segment whose starting block position is zero and its size is a single block. As a result, the size of E, $size(E)$, is given by

$$size(E) = 1 + \sum_{i=0}^{(\log_2 s)-1} size(seg[i]) = 1 + \sum_{i=0}^{(\log_2 s)-1} 2^i = s$$

The allocation status of E is denoted by two bits per segment: $00(0)$ for the free segment state, $11(1)$ for the allocated segment state, and $10/01(x)$ for the unavailable segment state.

Definition 3.2. *Given a small-size extent E, the allocation status of each segment is defined as follows:*

$$bit : seg[c] \rightarrow \{0, 1\}$$
$$bit : seg[i] \rightarrow \{0, 1, x\}, \quad 0 \leq i < \log_2 s$$

The extent where $bit(seg[c]) = 0$ is called the clean extent. If $bit(seg[c]) = 1$, then the extent is called the extent segment where it contains the remaining free spaces. After either the entire segments are filled with data or the time period for which the extent segment can stay at memory is expired, the extent is written to SSD partition.

Definition 3.3. *Let D be a set of segments of a small-size extent E allocated to a file.*

For any $seg[i] \in D$, $bit(seg[i]) = x$ *if* $\exists seg[h] \in D$ *such that* $h < i$ *and* $bit(seg[h]) = 1$.

The bit of a segment is marked as unavailable if the segment is not the starting one allocated to a file.

3.2.2. *Large-size extent.* For extents whose size is larger than a threshold δ (large-size extents), NF-hybrid creates sub-segments, in addition to segments described above. Those sub-segments are defined in the last segment, which is the largest one.

Definition 3.4. *Give a large-size extent L, the structure of L is defined as follows:*

$$L = \{seg[i] | c \leq i < \log_2 s\} \cup \{subseg[j] | 0 \leq j < (\log_2 s) - 1\}$$

where s is the size of L in blocks.

Figure 4 illustrates the structure of L. For segment i where $0 \leq i < \log_2 s$, the starting block position, $pos(seg[i])$, and the size in blocks, $size(seg[i])$, are both 2^i. On the other hand, for sub-segment j where $0 \leq j < (\log_2 s) - 1$, the start block position, $pos(subseg[j])$, and the size in blocks, $size(subseg[j])$, are $s/2 + 2^j$ and 2^j, respectively. As with the small-size extents, $pos(seg[c]) = 0$ and $size(seg[c]) = 1$. Therefore, the size of L is given by

$$\begin{aligned} size(L) &= 1 + \sum_{i=0}^{(\log_2 s)-2} size(seg[i]) + 1 + \sum_{j=0}^{(\log_2 s)-2} size(subseg[j]) \\ &= 2 \cdot \left(1 + \sum_{i=0}^{(\log_2 s)-2} 2^i\right) = s \end{aligned}$$

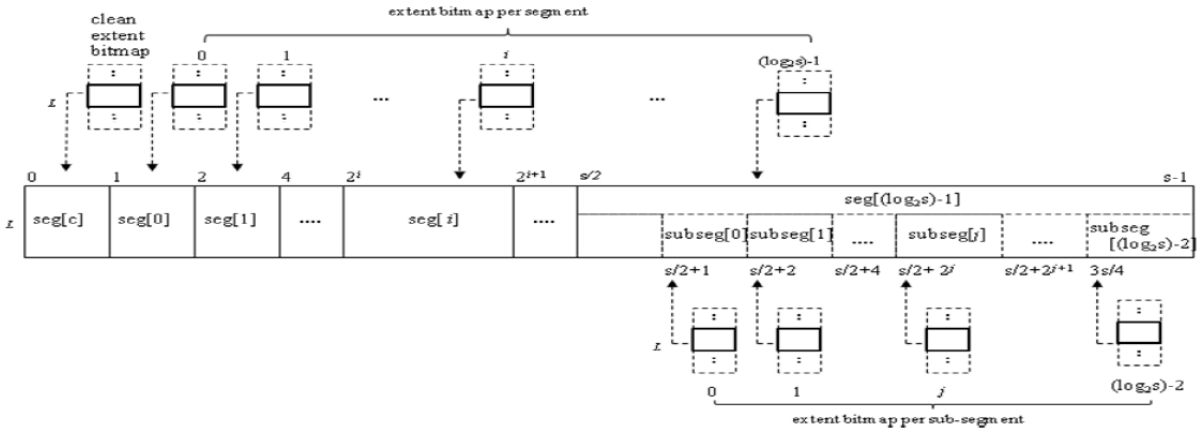

FIGURE 4. Large-size extent structure and bitmaps

Definition 3.5. *Let $seg[i]$ and $subseg[j]$ be segment i and sub-segment j, respectively, of a large-size extent L.*

$$\begin{aligned} &bit : seg[c] \rightarrow \{0, 1\} \\ &bit : seg[i] \rightarrow \{0, 1, x\},\ 0 \leq i < \log_2 s \\ &bit : subseg[j] \rightarrow \{0, 1, x\},\ 0 \leq j < (\log_2 s) - 1 \end{aligned}$$

In Definition 3.5, the same definition described in Definition 3.3 is applied for the bit value of x. In the data section composed of large-size extents, NF-hybrid creates $2 \log_2 s$ number of bitmaps: one for the clean extent, $\log_2 s$ bitmaps for segments and $(\log_2 s) - 1$ bitmaps for sub-segments. Assume that the size of data section is 32GB and the block size is 1KB. If a data section is configured with 16KB of extent size, then it requires five extent bitmaps, one for the clean extents and the other four bitmaps for the segments. The total space to store the extent bitmaps is $5\ 2^{19}$Bytes ($\sim$ 3MB). If the same data section is

Algorithm 1: *BITMAP* $(a, b, flag)$

1. find p and q such that $\lfloor 2^p \rfloor \leq a < 2^{p+1}$ and $\lfloor 2^q \rfloor \leq b < 2^{q+1}$ where $c \leq p$, $q < \log_2 s$;
2. **case** {
3. $p, q < (\log_2 s) - 1$: /* only segments are used for file operation */
4. $bit(seg[p]) \leftarrow 0$; /* free or allocate the first segment */
5. **if** $flag$ **then** $bit(seg[p]) \leftarrow 1$ **end if**
6. $segnum \leftarrow q - p$;
7. $p < (\log_2 s) - 1$, $q \geq (\log_2 s) - 1$: /* both segments and sub-segments are used */
8. $bit(seg[p]) \leftarrow 0$;
9. **if** $flag$ **then** $bit(seg[p]) \leftarrow 1$ **end if**
10. find q such that $\lfloor 2^q \rfloor \leq b - s/2 < 2^{q+1}$;
11. $segnum \leftarrow (\log_2 s) - (p + 1)$; $subsegnum \leftarrow q + 1$;
12. $p, q \geq (\log_2 s) - 1$: /* only sub-segments are used for file operation */
13. find q such that and $\lfloor 2^q \rfloor \leq b - s/2 < 2^{q+1}$;
14. **if** $a = pos(seg[(\log_2 s) - 1])$ **then**
15. $bit([seg[p]) \leftarrow 0$;
16. **if** $flag$ **then** $bit([seg[p]) \leftarrow 1$ **end if**
17. $subseqnum \leftarrow q + 1$; /* compute the remaining sub-segments */
18. **else**
19. find p such that $\lfloor 2^p \rfloor \leq a - s/2 < 2^{p+1}$;
20. $bit(subseg[p]) \leftarrow 0$;
21. **if** $flag$ **then** $bit(subseg[p]) \leftarrow 1$ **end if**
22. $subseqnum \leftarrow q - p$;
23. **end if**
24. }
25. **for** (remaining segments) **do**
26. $bit([seg[i]) \leftarrow 0$;
27. **if** $flag$ **then** $bit([seg[i]) \leftarrow x$ **end if**
28. **end for**
29. **for** (remaining sub-segments) **do**
30. $bit(subseg[j]) \leftarrow 0$;
31. **if** $flag$ **then** $bit(subseg[j]) \leftarrow x$ **end if**
32. **end for**

configured with 64KB of extent size, then it requires 12 extent bitmaps, one bitmap for the clean extents, six bitmaps for the segments and five bitmaps for the sub-segments. Therefore, the total space for the extent bitmaps is 12 2^{17}Bytes ($\sim$ 2MB).

Algorithm 1 shows the steps for partitioning extents, according to file allocation and deletion. If *BITMAP* is called for file allocation, then flag is set to *TRUE*. Otherwise, flag is passed as *FALSE* to *BITMAP*. In the algorithm, steps 1 to 24 calculate segments and sub-segments involved in the file operation. The time complexity for calculating segments and sub-segments is O(1). The steps 25 to 28 mark the remaining segments either as unavailable for file allocation, or as free for file deletion. Since there are $(\log_2 s)+1$ segments, the time complexity for marking segments is $O(\log_2 s)$. In the same way, the time complexity for marking sub-segments in steps 29 to 32 is $O(\log_2 s)$ because of $(\log_2 s) - 1$ sub-segments available in the extent. As a result, the time complexity of Algorithm 1 is $O(\log_2 s)$.

3.3. **In-memory extent table.** NF-hybrid performs file allocation using in-memory extent table. The extent table is constructed for each SSD data section at file system mount, by reading bitmaps. There are three main goals in using in-memory extent table: 1) to perform the in-memory extent bitmap operation, in order to improve the allocation time,

2) to minimize the extent fragmentation problem, in order to improve the utilization of SSD storage resources, and 3) to align data size with flash block boundaries.

In the extent table, table entry i connects the extents that have free space starting from segment i, except for the first table entry that is associated to clean extents. If a data section is composed of large-size extents, then the second extent table is created to allocate sub-segments. In this case, table entry j of the second extent table is associated to the extents that have free space starting from sub-segment j.

Each table entry has its own linked list of extent descriptors. The extent descriptor includes the information about the corresponding extent, such as extent address, total data size and number of segments mapped to files, and pointer to the callback function to be called 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, file and extent block positions, and mapping length.

Let *segTab* and *subsegTab* be the first and second extent tables, respectively. And let $d(L)$ and $hole[k]$ be the extent descriptor associated to an extent L and kth free space in L composed of segments and sub-segments. Finally, let $size(hole[k])$ and $pos(hole[k])$ be the size of $hole[k]$ in blocks and the block position of the starting segment/sub-segment of $hole[k]$. The $hole[l\arg e]$ denotes the largest free space.

Definition 3.6. *Let* $\Im = \{segTab = \{tb_0^i | c \leq i < \log_2 s\},\ subsegTab = \{tb_1^j | 0 \leq j < (\log_2 s) - 1\}\}$. *The structure of table entry is given by*

$$tb_0^c = \{d(L) | \forall L,\ bit(seg[c]) = 0\}$$

$$tb_0^i = \{d(L) | \forall L,\ bit(seg[i]) = 0,$$
$$pos(hole[l\arg e]) = pos(seg[i]),\ size(hole[l\arg e]) = \max\{size(hole[k]) | k \geq 0\}\}$$

$$tb_1^j = \{d(L) | \forall L,\ bit(subseg[j]) = 0,$$
$$pos(hole[l\arg e]) = pos(subseg[j]),\ size(hole[l\arg e]) = \max\{size(hole[k]) | k \geq 0\}\}$$

The movement of an extent between table entries occurs when the allocation process on the extent leaves the enough number of free blocks to be used. For example, in Figure 5, if the allocation process on a clean extent leaves free blocks from 2^i to the end, then its extent descriptor is linked to table entry i of the first extent table. If the extent is reused while leaving free space from sub-segment r, then the extent is moved to the second extent table and linked to table entry r. In NF-hybrid, an extent can stay at the extent table until either the entire blocks of the extent are used or the time period for which an extent can stay at the table is expired.

Algorithm 2 shows the steps in write operations for collecting segments or sub-segments in VFS layer using the extent table. Let f be a new file to be allocated in SSD partition

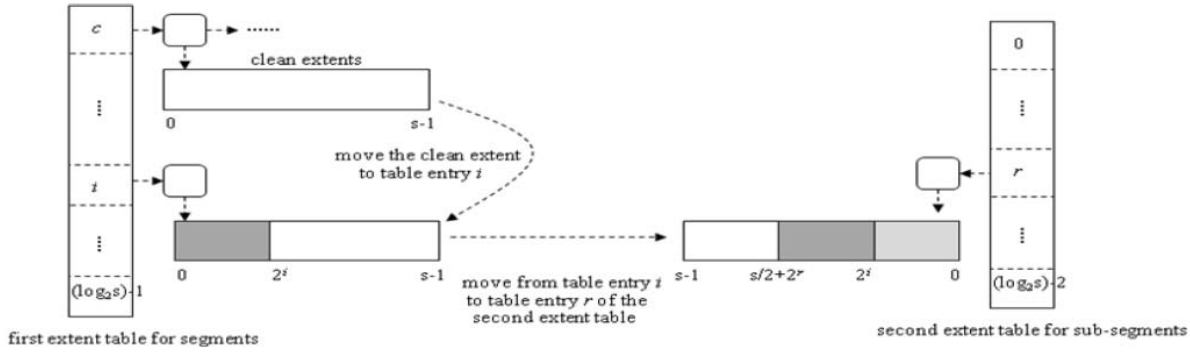

FIGURE 5. Extent movement

Algorithm 2: $MOVE\ (f, \Im)$

```
1.  if size(f) ≥ s then
2.      m = size(f)/s;
3.      /* let hole [0] be the unused space of the last extent */
4.      for m do
5.          remove d(L) from tb_0^c;
6.          if L is the last extent then BITMAP (0, pos(hole[0]) − 1, TRUE)
7.          else BITMAP (0, s − 1, TRUE) end if
8.      end for
9.      if pos(hole[0]) > s/2 then
10.         compute sub-segment r such that 2^r ≤ pos(hole[0]) − s/2 < 2^(r+1);
11.         link d(L) to tb_1^r, in the decreasing order of size(hole[0]);
12.     else
13.         compute segment r such that 2^r ≤ pos(hole[0]) < 2^(r+1);
14.         link d(L) to tb_0^r, in the decreasing order of size(hole[0]);
15.     end if
16. else
17.     for tb_0^i and tb_1^j do
18.         select L such that size(hole[l arg e]) ≥ size(f) and time(L) is the maximum;
19.     end for
20.     BITMAP (pos(hole[l arg e]), pos(hole[l arg e]) + size(f) − 1, TRUE);
21.     choose hole[l arg e] such that size(hole[l arg e]) = max{size(hole[k])|k ≥ 0};
22.     if pos(hole[l arg e]) > s/2 then
23.         compute r such that 2^r ≤ pos(hole[l arg e]) − s/2 < 2^(r+1);
24.         link d(L) to tb_1^r, in the decreasing order of size(hole[l arg e]);
25.     else
26.         compute r such that 2^r ≤ pos(hole[l arg e]) < 2^(r+1);
27.         link d(L) to tb_0^r, in the deceasing order of size(hole[l arg e]);
28.     end if
29. end if
```

and $time(L)$ be the time at which L is inserted into the extent table. In steps 4 to 8, the time for removing m clean extents from the first table entry takes $O(m)$. Since the time for performing *BITMAP* takes $O(\log_2 s)$, the time complexity for steps 4 to 8 is $O(m \log_2 s)$. In steps 9 to 15, the time for inserting the last extent to the appropriate linked list based on the largest unused space takes $O(n)$ where n is the number of elements of the linked list. Therefore, the time complexity for allocating a file whose size is no smaller than s is $O(n + m \log_2 s)$.

On the other hand, in steps 17 to 19, since there are $2 \log_2 s$ table entries total in two extent tables and the largest extents are located at the front, the time for selecting an extent for f takes $O(\log_2 s)$. In steps 22 to 28, searching for the appropriate position to insert L in the linked list of n elements takes $O(n)$. As a consequence, the time complexity of Algorithm 2 is $O(n + m \log_2 s)$.

3.4. **Mathematical analysis of allocation algorithm.** In this section, we formulize the allocation process of NF-hybrid.

Theorem 3.1. *Given an extent L, there is at least a sequence of allocation steps before L is written to SSD partition.*

Proof: We assume that, with an extent L composed of $n + 1$ segments the allocation process on L forms Markov chain. Let $S = \{st_i, st_{k_0}, st_{k_1}, \cdots, st_{k_m}, st_j : i \geq 0, j \leq n\}$ be a set of states where st_i implies that $i + 1$ segments starting from the first one are used

for an allocation. Furthermore, st_n implies the final state where all segments are used for the allocation process. Let $A = \{st_0, st_1, \cdots, st_{n-1}\}$ and $B = \{F|F = st_n\}$. Also, let p be the probability that an allocation process takes place and $q = 1 - p$. Especially, we define $p_{k_r k_{r+1}}$ as the Markov transition probability of from st_{k_r} to $st_{k_{r+1}}$. Also, let N_B be the minimum number of transitions arriving at the final state $st_j \in B$ from a state $st_i \in A$. Finally, we define β_i as the probability of reaching at $st_j \in B$ after performing a finite number of transitions from $st_i \in A$, which satisfies the followings:

$$\beta_i = \Pr(N_B < \infty | st_i) = \begin{cases} \sum\limits_{k \in A} p_{ik}\beta_j, & k \neq n \\ \sum\limits_{k \in B} p_{ik} = \Pr(N_B = a), & k = n, \quad 1 \leq a \leq n \end{cases}$$

Then, for a state $st_k \in A$, β_k is given by

$$\begin{aligned}
\beta_k &= \sum_{k \in S} p_{k_{i-1}k_i}\beta_k \\
&= \sum_{k_1 \in B} p_{k_0k_1} + \sum_{k_1 \in A} p_{k_0k_1}\beta_{k_1} \\
&= \Pr(N_B \leq 1) + \sum_{k_1 \in A} p_{k_0k_1}\beta_{k_1} \\
&= \Pr(N_B \leq 1) + \sum_{k_1 \in A} p_{k_0k_1}\left[\sum_{k_2 \in B} p_{k_1k_2} + \sum_{k_2 \in A} p_{k_1k_2}\beta_{k_2}\right] \\
&= \Pr(N_B \leq 1) + \sum_{k_1 \in A, k_2 \in B} p_{k_0k_1}p_{k_1k_2} + \sum_{k_1, k_2 \in A} p_{k_0k_1}p_{k_1k_2}\beta_{k_2} \\
&= \Pr(N_B \leq 2) + \sum_{k_1, k_2 \in A}\left[\prod_{i=0}^{1} p_{k_ik_{i+1}}\right]\beta_{k_2} \\
&= \Pr(N_B \leq 2) + \left[\sum_{k_1, k_2 \in A, k_3 \in B} p_{k_0k_1}p_{k_1k_2}p_{k_2k_3} + \sum_{k_1, k_2, k_3 \in A} p_{k_0k_1}p_{k_1k_2}p_{k_2k_3}\beta_{k_3}\right] \\
&= \Pr(N_B \leq 3) + \sum_{k_1, k_2, k_3 \in A}\left[\prod_{i=0}^{2} p_{k_ik_{i+1}}\right]\beta_{k_3} \\
&= \Pr(N_B \leq 3) + \cdots + \sum_{k_1, k_2, \cdots, k_{n-1} \in A, k_n \in B}\left[\prod_{i=0}^{n-1} p_{k_ik_{i+1}}\right] + \sum_{k_1, k_2, \cdots, k_n \in A}\left[\prod_{i=0}^{n-1} p_{k_ik_{i+1}}\right]\beta_{k_n} \\
&= \Pr(N_B \leq n) + \sum_{k_1, k_2, \cdots, k_n \in A}\left[\prod_{i=0}^{n-1} p_{k_ik_{i+1}}\right]\beta_{k_n}
\end{aligned}$$

Since there is a time limit that L can stay at the extent table,

$$\lim_{n \to \infty} \sum_{k_1, k_2, \cdots, k_n \in A}\left[\prod_{i=0}^{n-1} p_{k_ik_{i+1}}\right]\beta_{k_n} = 0.$$

Therefore,

$$\Pr(N_B \leq n) \approx \sum_{n=0}^{\infty} \Pr^{(n)} \tag{1}$$

We assume that $I_j(k)$ is the indicator function that, after k transitions, the Markov chain goes to state j:

$$I_j(k) = \begin{cases} 1, & \text{if } (j \in B) \\ 0, & \text{if } (j \notin B) \end{cases} \tag{2}$$

Let $M_n = \frac{1}{n} E\left[\sum_{k=0}^{n} I_j(k)\right] = \frac{1}{n}\sum_{k=0}^{n} \Pr^{(k)}$. Also, let π be the stationary probability of Markov chain [33]. By applying the Chebyshev inequality,

$$\Pr[|M_n - \pi| \le \varepsilon] \ge 1 - 1/k^2 \text{ where } \exists\varepsilon,\ k > 0 \quad \varepsilon = k \cdot \sqrt{pq/n}$$

$$1 \ge \Pr[|M_n - \pi| \le \varepsilon] \ge 1 - \frac{pq}{\varepsilon^2 n} \tag{3}$$

In Equation (3), $\lim_{n\to\infty} \frac{pq}{\varepsilon^2 n} = 0$ and $\lim_{n\to\infty} \Pr[|M_n - \pi| \le \varepsilon] = 1$. Also, according to the theorem of strong law of large numbers, $\Pr[\lim_{n\to\infty} M_n = \pi] = 1$ and thus $\lim_{n\to\infty} \frac{1}{n}\sum_{k=0}^{n} \Pr^{(k)} = \pi$. Because $\sum \pi = 1$, in Equation (1), $\sum_{n=0}^{\infty} \Pr^{(n)} \to 1$. As a result, after a number of transitions, there is at least one allocation sequence that arrives at F to be written to SSD partition.

3.5. **File mapping.** NF-hybrid supports the transparent file mapping in which files are allocated in the appropriate SSD data section, in order to effectively manage tight storage resources. The mapping information to the data section is determined by file access characteristics, such as file access pattern and usage. Furthermore, files can bypass SSD partition to be stored in only HDD partition. Bypassing is necessary to prevent SSD space from being consumed by files that do not need fast I/O service. Figure 6 illustrates the transparent file mapping. The file mapping is initialized at file system creation and, if necessary, can be modified at mount time. The information about the file mapping is stored in in-memory map table.

In Figure 6, the default data section D_0 is composed of extents with the size of δ blocks and is mapped to the directory */nfhybrid/usr* and the second data section D_1 composed of s blocks per extent is mapped to the directory */nfhybrid/data*. Since most files stored in */nfhybrid/streaming* have a large, sequential access pattern, the directory starting from */nfhybrid/streaming* is mapped to D_2, which is composed of the largest size of extents.

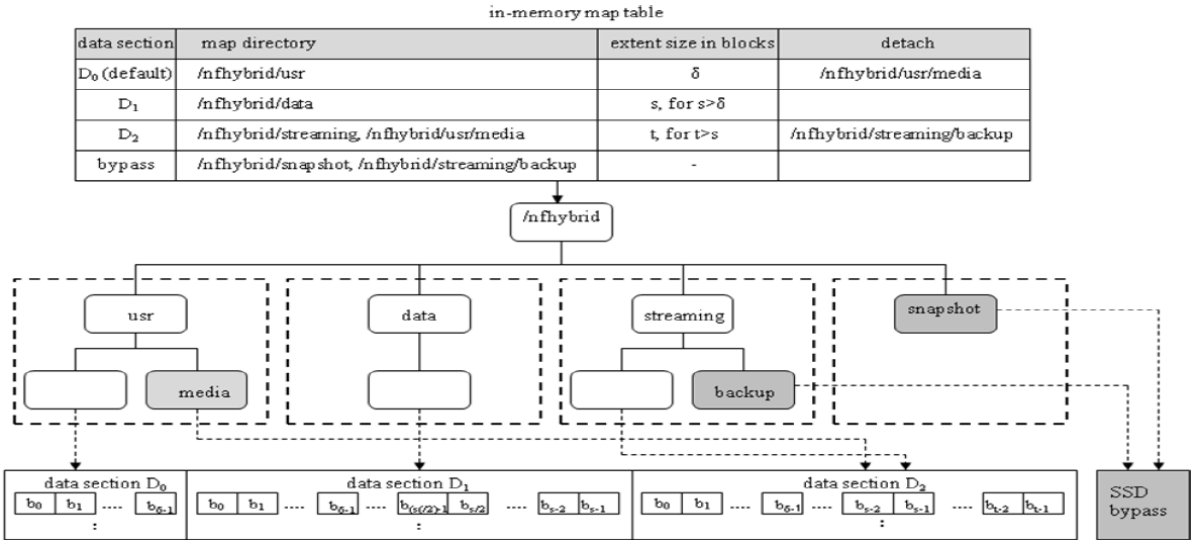

FIGURE 6. Transparent file mapping

Finally, since the files stored in */nfhybrid/snapshot* do not need high, interactive I/O response time, the directory path along with */nfhybrid/snapshot* is configured as *SSD bypass*. Once a directory is mapped to the data section, all the descendants in the same directory path will be mapped to the same data section.

When the access characteristics are changed, files can be remapped to another data section without involving any copy operations. For example, if a new sub-directory is created in the data section composed of small-size extents, to store files deploying a large, contiguous access pattern, then the mapping to the sub-directory can be moved to another data section consisted of a larger extent size. Figure 6 shows that */nfhybrid/usr/media* is remapped from D_0 to D_2, while retaining the same logical directory hierarchy. Also, the mapping for */nfhybrid/streaming/backup* is changed from D_2 to *SSD bypass*, because it is created to store only backup files.

In NF-hybrid, file write operation is simultaneously performed on both SSD and HDD address spaces. If either of write operations is completed, then control returns user. The file read operation is performed by checking *SSD_active* to see if the desire file is available in SSD address space. In case that *SSD_active* and *SSD_wr_done* are both enabled, the data stored in SSD address space are brought into memory. Otherwise, the data stored in HDD address space is brought into memory and also is updated to SSD address space.

The *SSD_active* and *SSD_wr_done* are also used for data recovery in SSD partition. NF-hybrid provides logs that are stored in both partitions. In SSD partition, a 4MB of log space is reserved, storing logs in a circular way. Also, NF-hybrid keeps the same logs in HDD partition, to use them in case of SSD crash. When data recovery takes place, NF-hybrid restores data by referring to *SSD_wr_done*. NF-hybrid checks the flag to see if the write operation has been completed before recovery. If SSD-related bit of the flag is not marked as one, then NF-hybrid frees the allocated SSD extents and turns off *SSD_active*. The data would be accessed later from HDD partition and be replicated to SSD partition.

To obtain the continuously available SSD space, NF-hybrid provides the extent replacement algorithm using a 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 head of the highest level.

The file eviction takes place when the available clean extents in the data section drop below a threshold. The extent replacement process begins by flushing out the files linked at the bottom queue and also releases the extents allocated to those files. Since 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.**

In this section, we describe I/O performance of NF-hybrid. The performance was compared with that of two other file systems, ext2 [30] and xfs [31], installed on both HDD and SSD. We choose ext2 for the performance comparison because HDD partition of NF-hybrid uses I/O kernel modules similar to those of ext2. Also, we choose xfs for the comparison because of its B+ tree-based extent allocation. Table 2 shows the experimental platform we used for the performance evaluation. For SSD partition, we installed a 80GB of fusion-io SSD ioDrive on the system. The operating system was CentOS release 5.5 with a 2.6.18 kernel.

4.1. **Experiments based on file access pattern.**

We used two benchmarks to evaluate I/O performance according to file access pattern: IOzone for sequential file operations and

TABLE 2. Experimental platform

processor	Intel Xeon CPU X5365, 3GHz, 4Core
memory	8*2048MB, Samsung M395T5750EZ4-CE66, 667MHz data transfer rate
L1 cache	128KB
L2 cache	8192KB
disk	750GB, 7200RPM, Seagate ST3750330AS
SSD	fusion-io ioDrive, 80GB, SLC, PCI-e interface, 50μs read access latencies

random I/O template for random file operations. Also, we called fsync() in every I/O operations, to minimize the effect of memory cache as much as possible. In IOzone, we used 4KB of record unit while varying file sizes from 64KB to 256MB where 4KB of data are sequentially executed in each file size.

In the random I/O template, files are accessed from the randomly chosen directories and the maximum depth of directories is ten. We changed files every 16KB of I/O, to maximize the effect of random I/O accesses. In the evaluation, we will see how significantly the mechanical moving overhead of HDD and the semiconductor overhead of SSD affect I/O performance, especially for files randomly located at directory hierarchies.

Figure 7 shows the write performance of NF-hybrid, comparing with that of ext2 and xfs installed on HDD and SSD. In the evaluation, NF-hybrid configures the entire SSD partition as 32GB of a single data section composed of 16KB of extent size. Since the block size is set to 1KB, NF-hybrid creates five bitmaps: one bitmap for the clean extents and the other four bitmaps for the extent segments.

In Figure 7, with 256MB of files, the write bandwidth of NF-hybrid is almost four times higher than that of ext2 and xfs installed on HDDs. According to the evaluation, we can observe that most of files are written to SSD partition in NF-hybrid due to its temporal locality. When compared with the write performance of ext2 and xfs installed on SSDs, NF-hybrid produces almost the same bandwidth to that of both file systems. However, due to its hybrid device structure being integrated with HDD partition, NF-hybrid can offer the larger storage capacity than those file systems installed on SSDs.

In Figure 8, we compared the rewrite performance of NF-hybrid with that of both ext2 and xfs. Similar to write measurement on HDD, the rewrite throughput of NF-hybrid

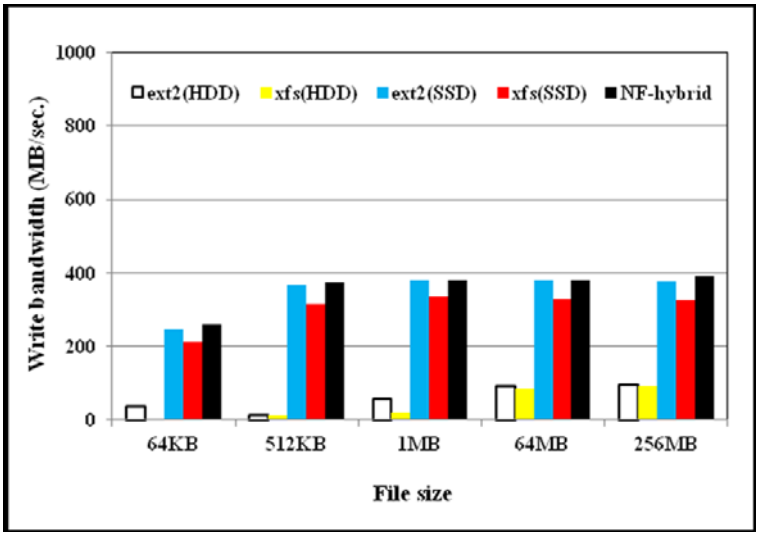

FIGURE 7. IOzone write

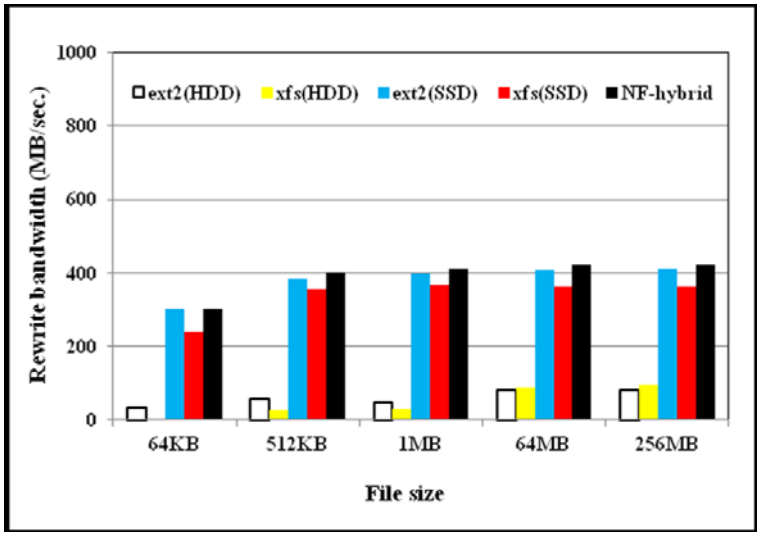

FIGURE 8. IOzone rewrite

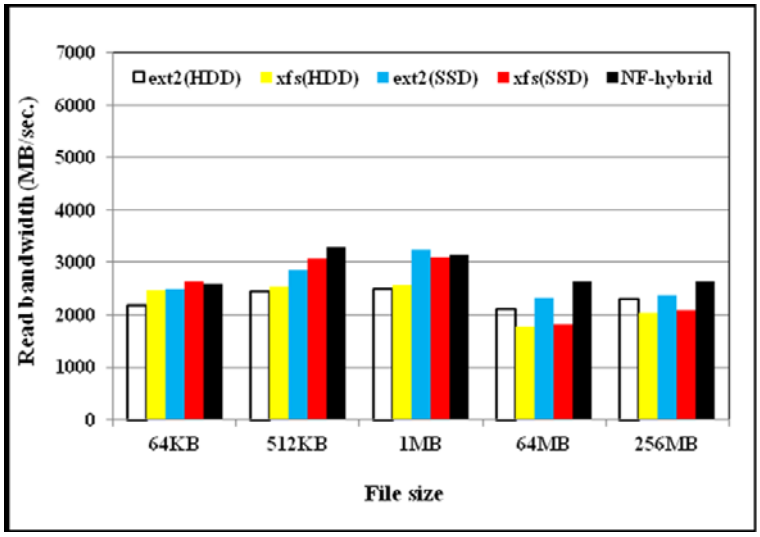

FIGURE 9. IOzone read

outperforms that of both ext2 and xfs installed on HDDs. When compared with that of both file systems on SSDs, NF-hybrid generates almost the same bandwidth with ext2, but is marginally faster than xfs.

The read performance of NF-hybrid is illustrated in Figure 9, while comparing it with the read performance of ext2 and xfs installed on two devices. In the read experiment, we can see that memory cache affects I/O performance to a large degree. In this case, NF-hybrid does not produce the significant performance difference, when compared with that of ext2 and xfs installed on HDDs. We guess that the prefetching scheme implemented in ext2 and xfs might compensate the performance difference between two devices. As shown in the write experiment, NF-hybrid generates the similar performance bandwidth to that of two other file systems installed on SSDs, thus proving the effectiveness of its hybrid internal structure.

In Figures 10 to 12, we evaluated I/O performance using the random I/O template. This experiment was performed to observe I/O behavior of file systems, with little impact of memory cache. Figure 10 shows the write performance of three file systems. In case of 256MB of file size, the performance difference between NF-hybrid and ext2 and xfs

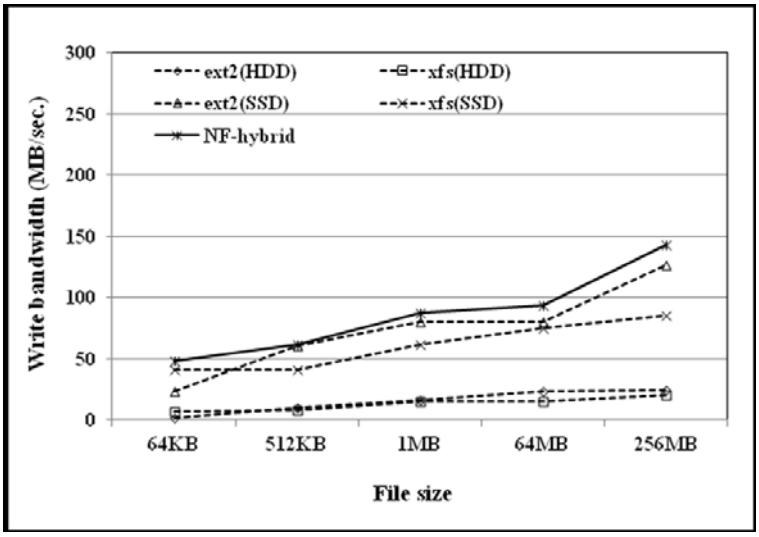

FIGURE 10. Random write

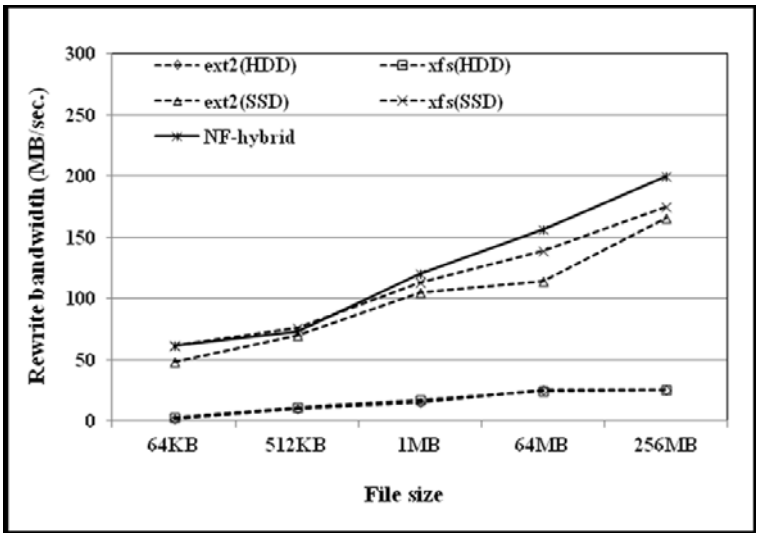

FIGURE 11. Random rewrite

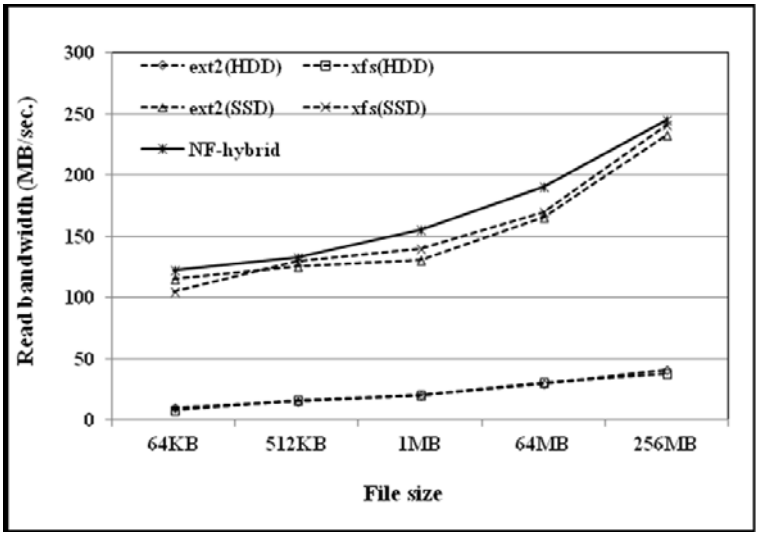

FIGURE 12. Random read

installed on HDDs is higher than that shown in Figure 7. Such a difference takes place because most of files of NF-hybrid are accessed from SSD partition, which rarely produces the positioning overhead in accessing randomly located files.

However, NF-hybrid shows a slight performance decrement at the midway because of the bitmap operation in the extent table. NF-hybrid refers the extent table to allocate the clean extents and marks the associated bits in the clean extent bitmap. Also, the bitmaps should be written to SSD partition periodically. These operations decrease the write performance at some point. Although such an overhead produces the performance turbulence at the midway, it does not increase along with file sizes because we can notice the performance speedup as file sizes increase. As a result, in the applications where a lot of randomly located files are accessed, NF-hybrid offers good I/O bandwidth, while providing the large storage capacity.

Figure 11 illustrates the rewrite performance of three file systems. Until 512KB of file size, there is no significant performance difference between NF-hybrid and ext2 installed on SSD. However, from 1MB of file size, the performance of NF-hybrid is slightly higher than that of the others, due to the large I/O granularity. We guess that the allocation process using the in-memory extent table and data alignment to flash block boundaries might cause such a performance increment. Also, the less number of bitmap updates than that of write operations contributes to generate better I/O bandwidth.

Figure 12 shows the read bandwidth of the random I/O template. In this experiment, the read behavior of NF-hybrid differs from one observed in Figure 9 where no significant difference is found between the performances of NF-hybrid and two other file systems installed on HDDs. This is because we cannot expect the effect of prefetching scheme in ext2 and xfs in the random I/O template. Consequently, in random I/O experiments, the strength of NF-hybrid is more obvious than that in sequential experiments of IOzone benchmark.

4.2. **Experiments based on extent size.** In this section, we present the experiment with various extent sizes. In NF-hybrid, the extent size significantly affects I/O performance because the bitmap and extent structures are closely related to the extent size. In this experiment, we created four 16GB of data sections whose extent size is configured as 16KB, 64KB, 256KB and 512KB each. Those data sections are mapped to */nfhybrid/test16*, */nfhybrid/test64*, */nfhybrid/test256* and */nfhybrid/512*, respectively. On each data section, we varied file sizes from 64KB to 256MB, to notice the effect of the flexible data layout. In SSD partition, the extent sizes larger than 16KB create two kinds of bitmaps: one for each segment and the other for each sub-segment. Table 3 shows the number of bitmaps and storage requirement for bitmaps, according to the extent size.

Figures 13 to 15 show I/O performances using IOzone, while varying extent size. In read operations, although we use fsync() option, we can see that memory cache still affects the read bandwidth. In Figure 13, with 64KB of file size, we can notice that the write performance of using 512KB of extent size is a little slower than that of using 16KB of extent size. This is because if the file size of a new file is smaller than the extent size, then a single extent is assigned to the file and should wait at the extent table until the

TABLE 3. The number of bitmaps and storage requirement per extent size

extent size	16KB	64KB		256KB		512KB	
bitmap category	per seg.	per seg.	per sub-seg.	per seg.	per sub-seg.	per seg.	per sub-seg.
no. of bitmaps	5	7	5	9	7	10	8
storage space	$\sim$ 2MB	768KB		256KB		144KB	

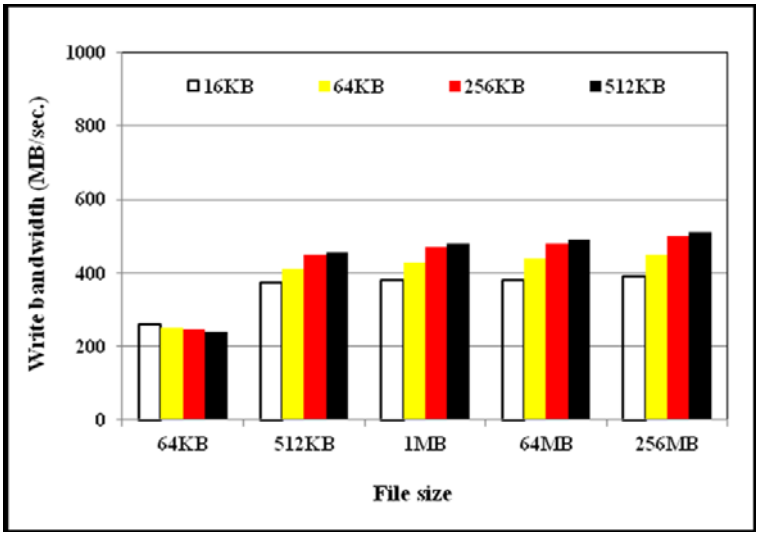

FIGURE 13. IOzone write

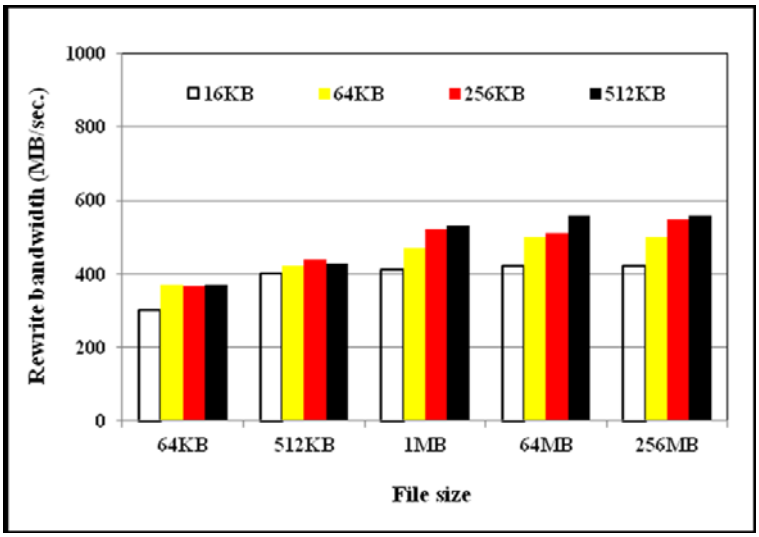

FIGURE 14. IOzone rewrite

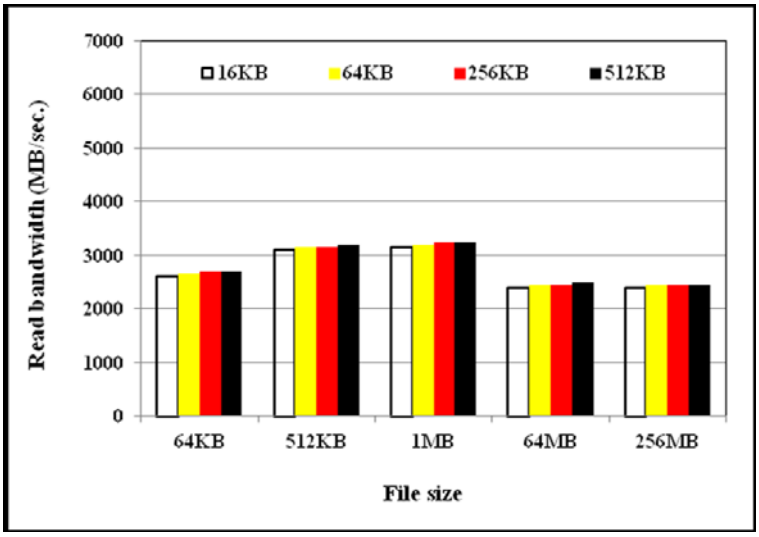

FIGURE 15. IOzone read

extent is occupied by other files. When the write delay at the extent table exceeds a time limit, then the extent is written to SSD partition without being fully occupied.

This observation tells us that the extent size of a data section should carefully be defined, to reduce such an overhead in NF-hybrid. One efficient way of defining the extent size is that the directory hierarchy frequently being accessed for file updates is mapped to the data section with small-size extents and the files whose data are accessed in a large granularity are mapped to the data section with large-size extents. In this way, we could obtain the advantage of the transparent file mapping, which allows mapping data sections to different directory hierarchies, according to file access characteristics.

However, as file size increases, using large-size extents produces more performance speedup. For example, in writing 256MB of file size, the write performance with 512KB of extent size is 23% higher than that with 16KB of extent size. This is because the large-size extents need the less number of extents to be allocated than the small-size extents do. The interesting point is that no significant performance difference can be found between 256KB and 512KB of extent sizes. We guess that the buffer of I/O controller might be a bottleneck in generating higher speedup with the larger extent size.

Figure 14 illustrates the rewrite performance in which we could find the similar performance behavior to that of the write performance. In this experiment, we overwrite files so that no bitmap operation is required. However, the extent should be filled with the new data using the extent table. Similar to the write performance, we could observe the performance speedup from 1MB of file size with 64KB of extent size because the less number of extents is used for the rewrite operations.

In the read performance (Figure 15), changing the extent size does not produce the significant performance difference. There are two reasons for such a read behavior. First, accessing the large number of extents from SSD partition does not incur the moving overhead to positioning desire data. When a large-size file is allocated using small-size extents, it needs more extents than that of using large-size extents. However, due to the absence of moving overhead, SSD partition does not produce the large performance difference between them. Second, the read path does not use the extent table and bitmaps, which are affected by the extent size. Additionally, the effect of memory cache might compensate the performance difference. As a result, using the different extent size for accessing a file has little impact on the read performance.

We evaluated the effect of the extent size, while accessing randomly located files in the directory hierarchy. We used the same random I/O template for this evaluation. Figure 16 shows the write performance, while varying the extent size for each file size. In case of random file accesses, I/O advantage using large-size extents is obvious as shown in Figure 16. For example, in writing 256MB of files, using 512KB of extent size produces 28% higher performance benefit than using 16KB of extent size.

Although SSD does little generate the positioning overhead for random accesses, writing random files produces more I/O overhead than writing sequential files since it needs more processes for performing flash-specific operations, such as block mapping considering wear-leveling. In this case, we guess that NF-hybrid gives an opportunity to reduce such an overhead by converting small-size random accesses to large-granularity accesses. The same performance gain can be observed in the rewrite performance illustrated in Figure 17 where large-size extents for randomly located files achieve higher bandwidth than small-size extents.

In case of read operations, like shown in Figure 18, large-size extents do not offer the significant performance gain because read operations do not need to go though the extent table in NF-hybrid. However, because the large-size extents require the less number of

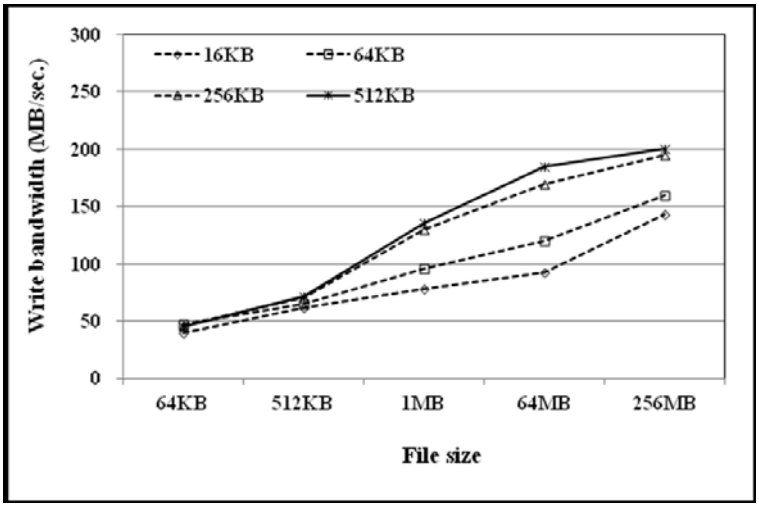

FIGURE 16. Random write

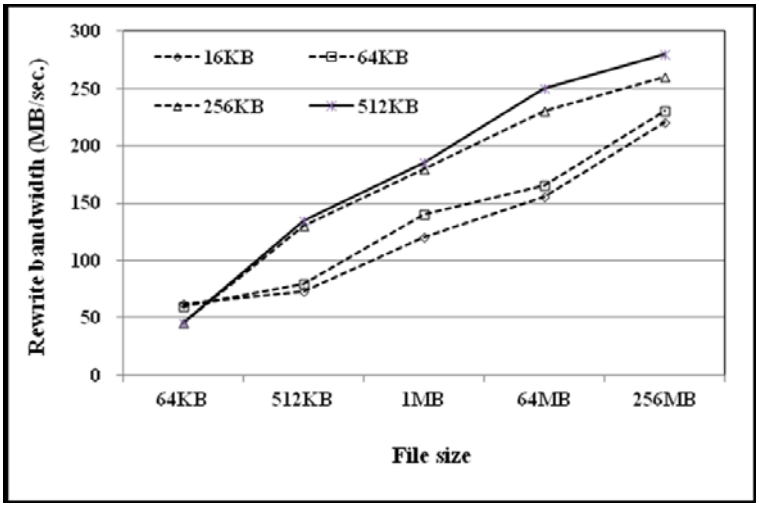

FIGURE 17. Random rewrite

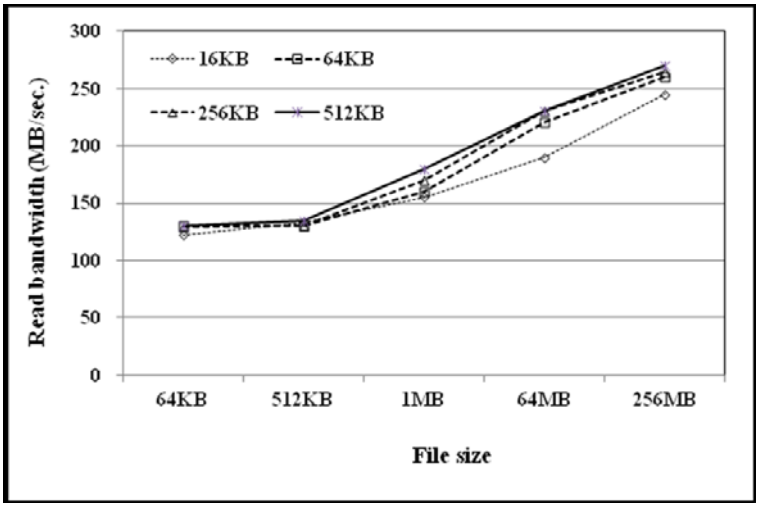

FIGURE 18. Random read

accesses than the small-size extents do, we could observe the slight performance improvement with the large-size extents.

4.3. **Experiments based on file size.** We used Postmark [38] to evaluate NF-hybrid with small-size files and used Bonnie++ [39] to evaluate with large-size files. Postmark is used to measure I/O performance on small-file system workloads, such as electronic mail and netnews services. In our experiment, we created files whose size was between 500B and 10KB by running 100,000 transactions. The number of files is varied from 1,000 to 20,000 and the ratio of read to append operations is set to 5:5 with which two operations equally likely take place. Bonnie++ is another public benchmark to measure hard drive and file system performance. We present Bonnie++ sequential I/O performance using 16KB of chunks, on top of 1GB of file. We used $-b$ option to invoke fsync() for write and rewrite operations.

Figure 19 shows the transaction rates per second ($\times 1000$) for ext2, xfs and NF-hybrid. As the number of files increases, the transaction rate becomes small due to the reduction of the available 128 byte-inodes in the directory. Both ext2 and xfs are installed on HDD and SSD. Also, the same data layout described in Section 4.2 was used for NF-hybrid configuration, where four 16GB of data sections were created using from 16KB to 512KB of extent size each. Those data sections are labeled from NF-hybrid (16KB) to NF-hybrid (512KB). In the figure, no significant performance difference shows between ext2 on HDD and ext2 on SSD. Although SSD rarely produces the moving overhead to position data, writing a large number of small-size files might cause significant semiconductor overhead on FTL layer to perform erase and write operations. Such an overhead degrades I/O performance as much as the mechanical moving overhead on HDD does to write small-size files.

As can be seen in Figure 19, NF-hybrid can suggest a solution in which small-size files are converted into large I/O granularities in VFS layer prior to write operations to SSD partition. For example, with 1000 and 20,000 files, the transaction rate of NF-hybrid (512KB) is 18% and 28% higher than that of ext2 installed on SSD, respectively. Even though NF-hybrid (512KB) causes the overhead to collect data in VFS layer, it does not greatly decrease I/O performance because its transaction rate is still 20% higher than that of NF-hybrid (16KB), with 20,000 files.

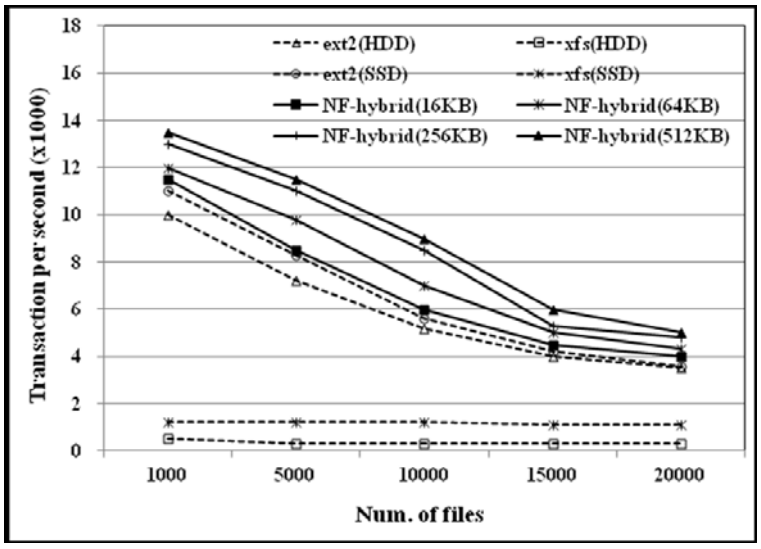

FIGURE 19. Transaction rates with Postmark

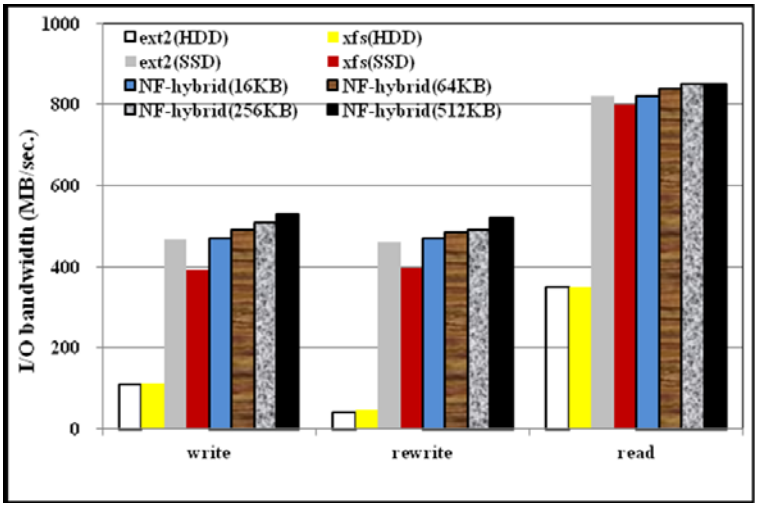

FIGURE 20. Bonnie++ I/O bandwidth

In Figure 20, in case of write operations, the performance difference between ext2 on HDD and SSD is 76%, due to the sequential access pattern in a large file. Even in this case, prior to write operations to SSD partition, collecting data in VFS layer to convert into a large I/O granularity can contribute to improve I/O performance since we can see that NF-hybrid (512KB) produces 11% higher write throughput over ext2 installed on SSD. On the other hand, reading with a large I/O granularity does not show the same performance improvement because the read operation does not cause the semiconductor overhead in SSD partition. However, reading with a large I/O granularity requires the less number of accesses than ext2 on SSD, resulting in a little performance increment in NF-hybrid (512KB).

5. **Conclusion.** In this paper, we presented NF-hybrid file system, which has been implemented on top of the hybrid structure. The main goal of NF-hybrid is to exploit the device characteristics of both HDD and SSD, to build large-scale storage subsystems in a cost-effective way. The integration of SSD partition is performed in three aspects. First, NF-hybrid uses SSD partition as a write-through persistent cache, to utilize the performance potentials of SSD. Second, NF-hybrid enables to construct multiple, logical data sections on SSD partition in such a way that their extent size can differently be defined each. Such a scheme helps to manage tight SSD storage resources efficiently, because it allows files to be assigned to the proper data sections in terms of file access characteristics. Third, on top of VFS layer, NF-hybrid attempts to mitigate the overhead of extent fragmentation and FTL bottleneck of doing erasure operation. To optimize the overhead, the extents of NF-hybrid are partitioned into segments or sub-segments while their size can be aligned with flash block boundaries, using extent bitmaps and in-memory extent table. We conducted the performance evaluation of NF-hybrid while comparing with that of ext2 and xfs installed on SSD and HDD. The experiments show that NF-hybrid produces higher performance than ext2 and xfs installed on HDDs. Such a performance speedup is especially apparent in write and rewrite operations, despite I/O access pattern. In case of read operations, the sequential read throughput of NF-hybrid with IOzone benchmark does not produce the significant improvement over ext2 and xfs installed on HDDs, due to the impact of memory cache. However, with the random I/O template, NF-hybrid shows better read performance than ext2 and xfs installed on HDDs. We also notice that NF-hybrid can offer the comparable I/O performance to file systems installed on SSDs.

However, since NF-hybrid is integrated with HDD, it can offer the much larger storage capacity than those file systems. The extent size also affects I/O throughput of NF-hybrid. Especially, file write and rewrite operations show the performance improvement with large-size extents because of data conversion to the large I/O granularity. Finally, we used Postmark and Bonnie++ to measure NF-hybrid with small-size files and large-size files, respectively. The experiment with Postmark shows that, despite SSD's strength of rarely generating mechanical moving parts, writing a large number of small-size files on SSD partition degrades I/O performance due to the significant semiconductor overhead for erasing flash blocks. In such a case, collecting data in VFS layer before passing to SSD can be a solution to improve I/O performance. Even in Bonnie++ experiment for large-size files, data conversion into a large I/O granularity in VFS layer can contribute to increasing I/O bandwidth. As a future work, we will evaluate NF-hybrid with real applications to verify its effectiveness and suitability.

Acknowledgment. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0025935) and (2011-0003662).

REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse and R. Panigrahy, Design tradeoffs for SSD performance, *Proc. of USENIX Annual Technical Conference*, Boston, MA, pp.57-70, 2008.

[2] A. Rajimwale, V. Prabhakaran and J. D. Davis, Block management in solid-state devices, *Proc. of USENIX Annual Technical Conference*, San Diego, CA, 2009.

[3] T. Sato and S. Masuda, Track-seeking control of a hard disk drive using multirate generalized predictive control to improve intersample performance, *International Journal of Innovative Computing, Information and Control*, vol.5, no.12(A), pp.4513-4522, 2009.

[4] 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*, vol.57, no.7, pp.1002-1008, 2008.

[5] G. Soundararajan, V. Prabhakaran, M. Balakrishnan and T. Wobber, Extending SSD lifetimes with disk-based write caches, *Proc. of the 8th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2010.

[6] J. Hsieh, L. Chang and T. Kuo, Efficient identification of hot data for flash-memory storage systems, *ACM Transactions on Storage*, vol.2, no.1, pp.22-40, 2006.

[7] H. Kim, E. Nam, K. Choi, Y. Seong, J. Choi and S. Min, Development platforms for flash memory solid state disks, *Proc. of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, Orlando, Florida, 2008.

[8] H. Kim and S. Ahn, BPLRU: A buffer management scheme for improving random writes in flash storage, *Proc. of the 6th USENIX Symposium on File and Storage Technologies*, San Jose, CA, 2008.

[9] I. Doh, J. Choi, D. Lee and S. Noh, Exploiting non-volatile RAM to enhance flash file system performance, *Proc. of the 7th ACM/IEEE International Conference on Embedded Software*, Salzburg, Austria, pp.164-173, 2008.

[10] L. Chang and T. Kuo, Efficient management scheme for large-scale flash-memory storage systems with resource conservation, *ACM Transactions on Storage*, vol.1, no.4, pp.381-418, 2005.

[11] A. Olson and D. Langlois, Solid state drives – Data reliability and lifetime, *White Paper, Imation Corporation*, pp.1-27, 2008.

[12] C. Park, W. Cheon, J. Kang, K. Roh and W. Cho, A reconfigurable FTL (Flash Translation Layer) architecture for NAND flash-based applications, *ACM Transactions on Embedded Computing Systems*, vol.7, no.4, pp.1-23, 2008.

[13] J. Kim, J. Kim, S. Noh, S. Min and Y. Cho, A space-efficient flash translation layer for compactflash systems, *IEEE Transactions on Consumer Electronics*, vol.48, no.2, pp.366-375, 2002.

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

[15] M. Saxena and M. Swift, FlashVM: Virtual memory management on flash, *Proc. of USENIX Annual Technical Conference*, Boston, MA, 2010.

[16] IBM Corporation, An assessment of SSD performance in the IBM system storage DS 8000, *IBM Technical Report*, 2009.
[17] *Solid-State Drive Comparison Chart, News & Reviews*, http://ssd-reviews.com.
[18] M. Polte, J. Simsa and G. Gibson, Comparing performance of solid state devices and mechanical disks, *Proc. of the 3rd Petascale Data Storage Workshop Held in conjunction with Supercomputing*, Austin, Texas, 2008.
[19] Samsung Electronics, K9XXG08XXM flash memory, *Technical Report*, 2007.
[20] Fusion-io, IoDrive user guide for Linux, *Technical Report*, 2009.
[21] L. Chang and C. Du, Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers, *ACM Transactions on Design Automation of Electronic Systems*, vol.15, no.1, pp.1-36, 2009.
[22] 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*, vol.2, no.3, pp.1-33, 2006.
[23] Z. Zhang and K. Ghose, hFS: A hybrid file system prototype for improving small file and metadata performance, *Proc. of EuroSys*, Lisboa, Portugal, 2007.
[24] 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*, vol.6, no.1, pp.1-25, 2010.
[25] D. Woodhouse, JFFS: The journaling flash file system, *Proc. of the Ottawa Linux Symposium*, Ottawa, Canada, 2001.
[26] E. Gal and S. Toledo, A transactional flash file system for microcontrollers, *Proc. of USENIX Annual Technical Conference*, Anaheim, CA, 2005.
[27] E. Gal and S. Toledo, Algorithms and data structures for flash memories, *ACM Computing Surveys (CSUR)*, vol.37, no.2, pp.1-30, 2005.
[28] H. Dai, M. Neufeld and R. Han, ELF: An efficient log-structured flash file system for micro sensor nodes, *Proc. of SenSys*, Baltimore, MA, 2004.
[29] M. Rosenblum and J. Ousterhout, The design and implementation of a log-structured file system, *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, pp.1-15, 1991.
[30] R. Card, T. Ts'o and S. Tweedie, Design and implementation of the second extended filesystem, *Proc. of the 1st Dutch International Symposium on Linux*, Seattle, WA, 1994.
[31] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Peck, Scalability in the XFS file system, *Proc. of USENIX Technical Conference*, San Diego, CA, 1996.
[32] *YAFFS (Yet Another Flash File System)*, http://www.yaffs.net.
[33] A. Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering*, Pearson Education, New Jersey, 2009.
[34] S. Park, D. Jung, J. Kang, J. Kim and J. Lee, CFLRU: A replacement algorithm for flash memory, *Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Seoul, Korea, 2006.
[35] 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*, vol.54, no.3, pp.1215-1223, 2008.
[36] 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*, vol.52, no.2, pp.485-493, 2006.
[37] 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*, vol.6, no.3, pp.1-27, 2007.
[38] J. Katcher, PostMark: A new file system benchmark, *Technical Report 3022*, http://www.netapp.com/technology/level3.
[39] *Bonnie++1.03a*, www.coker.com.au/bonnie++.