Extent-Based Allocation Scheme for Hybrid Storage Solutions

Jaechun No
College of Electronics and Information Engineering
Sejong University
Seoul, Korea
email:jano@sejong.ac.kr

Sung-soon Park
Dept. of Computer Science and Engineering
Anyang University and Gluesys Co. LTD
Anyang, Korea
email:sspark@gluesys.com

***Abstract*—We present an extent-based allocation scheme for hybrid storage solutions, called MatBall (Matrix and extent-based allocation), whose objective is to increase space utilization of SSD (Solid State Device) partition in the hybrid file system by reducing fragmentation overhead. In MatBall, to consume the remaining spaces as much as possible posterior to file allocations, I/O units (extents) of the hybrid file system are recursively partitioned into segments in the subsequent level and further file allocations are performed in units of the partitioned segments. Since MatBall defines easy-to-compute segment sizes and block positions in I/O units, allocating more files in the remaining spaces can be performed with a little overhead. The performance measurement with IOzone shows that the hybrid file system using MatBall enables to produce higher bandwidth over ext2 installed on HDD (Hard Disk Drive) and SSD.**

Keywords-extent partitioning; matrix-based allocation; file mapping; fragmentation overhead.

I. INTRODUCTION

SSD [1]-[4] technology has dramatically improved over decades to become an essential component in storage solutions. Due to the fact that SSD does not need the mechanical overhead, such as seek time, to locate the desire data, it has drawn great attention from IT markets that seek for improved I/O performance. The key obstacle to the widening SSD adoption to large-scale storage subsystems is its high cost per capacity, compared to that of HDD. Even though the cost of flash memory becomes decrease, the price of SSD is still much higher and such a high cost/capacity ratio makes it less desirable to construct large-scale storage subsystems solely composed of SSD devices.

There are several ways of utilizing SSD advantages to boost I/O performance [5]-[8][12]. The first one is to implement SSD-related I/O optimizations in the file system level. For example, Josephson et al. [6] uses fusion-io ioDrive to provide the virtualization flash storage layer that acts as the traditional block device driver with FTL (Flash Translation Layer). Also, Lee et al. [7] proposed a new filesystem metadata platform that can reduce SSD-specific semiconductor overheads.

Although those file systems have successfully integrated SSDs to improve I/O performance, adapting a new file system to the existing storage solutions is not easy because it should go through the long, pains-taking process to prove the durable data consistency and reliability.

An alternative is to use hybrid storage subsystems, which are managed by the hybrid file system or SSD-specific cache [9-11]. In such methods, a small portion of SSD partition is combined with a much larger HDD storage capacity in a cost-effective way, while making use of the strengths of both devices. Since only a small-size of file system address space is provided by SSD partition, increasing space utilization for SSD partition has a critical impact in improving I/O performance.

In this paper, we propose an extent-based file allocation approach, called MatBall. The primary objective of MatBall is to increase the usage of the costly SSD storage resources as much as possible in the hybrid file system, by reusing the remaining spaces of I/O units and thus by decreasing fragmentation overhead. In the hybrid file system where the entire address spaces are constructed on both SSD and HDD partitions, MatBall can contribute to maximize the space usage of SSD partition by taking responsibility of allocating files in SSD partition.

The rest of paper is organized as follows: In Section II, we present the implementation details of MatBall. The performance results of MatBall integrated with the hybrid file system are shown in Section III. In Section IV, we conclude our paper.

II. IMPLEMENTATION DETAILS

We present the segment partitioning and file mapping.

A. System Model

In MatBall, I/O unit is an extent. However, an extent is composed of a group of segments and the allocation on the extent is performed in units of segments to reduce extent fragmentation.

Definition 1 (extent structure) An extent of size s in blocks is a finite set of segments such that: 1) there are $\log_2 s+1$ number of segments at the top level (level 0), with each being indexed from H to $(\log_2 s)-1$; 2) segment j at level L whose size is larger than or equal to a threshold λ is partitioned into $j+1$ segments at the subsequent level $L+1$ and their indices are ranged from H to $j-1$. The segment partitioning is continued until the size of every segment is smaller than λ.

The segment with index H is called the head segment.

The segment *j* of level *L* being partitioned from segment *i* of level *L*-1 is the child and denoted as *seg*[*i*,*k*]. On the other hand, segment *t* of level 0 is denoted as *seg*[*,*t*].

The starting block position of segment *j* of level *L* is *pos*(*seg*[*i*,*j*])=*pos*(*seg*[*i*,*H*])+2^j where *j*>*H*. If *j*=*H*, then *pos*(*seg*[*i*,*j*]) is the same to the starting block position of segment *i*. The size of segment *j* is 2^j, except for the head segment that is composed of a single block.

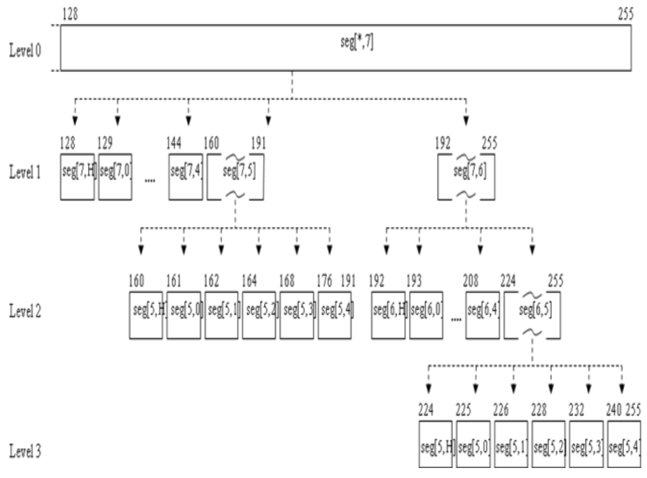

Figure 1. An example of segment partitioning.

Figure 1 illustrates an example of segment partitioning with λ =32. An extent with 256 blocks is partitioned into nine segments from *seg*[*,*H*] to *seg*[*,7] at the top level. Since the threshold is set to 32, the segments whose size is larger than or equal to 32 in blocks are partitioned at the subsequent level until each segment has the size of less than 32 blocks. In Figure 1, *seg*[*,7] is split into eight segments at level one and segments *seg*[7,5] and *seg*[7,6] are in turn partitioned at level two due to their sizes. The maximum segment index is decreased by one, in case the segment partitioning takes place at the subsequent level.

B. Allocation Matrix

The segment partitioning of MatBall takes place using the allocation matrix, which is organized at each level.

Definition 2 (Allocation matrix) An allocation matrix $\Phi^L = x[N+1, N]$ where *N*=log₂*s* is an abstraction of the segment partitioning at level *L*>0 such that: 1) each row *i* $(H \le i < \log_2 s)$ shows the segment index of the parent at level *L*-1; 2) column *j* $(H \le j < (\log_2 s) - 1)$ shows the index of segment at level *L* that is partitioned from parent *i*.

Figure 2(a) shows the allocation matrix Φ^L where $N = \log_2 s$ and $i = \log_2 \lambda$. There are two aspects in the allocation matrix. First of all, the segments from *seg*[*i*+1,*i*] to *seg*[*N*-1,*N*-2] should be partitioned at level *L*+1 since their size is larger than equal to λ. Second, some of them can have the same indices at the subsequent level. For example, segments from *seg*[*i*+1,*i*] to *seg*[*N*-1,*i*] contain *seg*[*i*,*H*] to *seg*[*i*,*i*-1] at level *L*+1. In Φ^L, *x*[*i*,*j*] is the number of segments with index *j* at level *L* that are partitioned from the segments with index *i* at level *L*-1. If *x*[*i*,*j*]=0 for row *i*, then no segment partitioning takes place at level *L*.

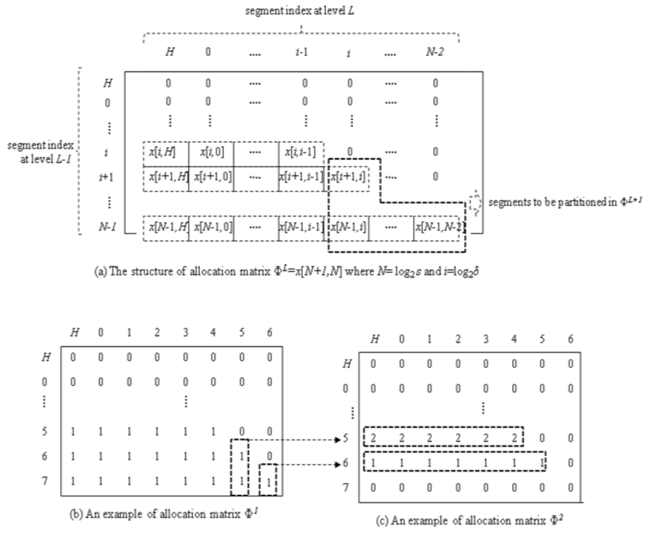

Figure 2. An example of allocation matrix.

Figure 2(b) and (c) show the allocation matrix Φ^1 and Φ^2 for an extent of size 256 blocks. The allocation matrix consists of nine rows and eight columns. The rows of Φ^1 denote the segments of the top level from *seg*[*,*H*] to *seg*[*,7]. Among them, *seg*[*,5] to *seg*[*,7] are partitioned at level one because their sizes are at least λ. Also, the children of *seg*[6,5] and *seg*[7,5] contain *seg*[5,H] to *seg*[5,4] at level two and thus *x*[5,*H*] to *x*[5,4] of Φ^2 are marked as two. On the other hand, *x*[6,*H*] to *x*[6,5] are set to one because only *seg*[7,6] of Φ^1 is involved in the segment partitioning. Since *x*[6,5]>0 in Φ^2, one more partitioning would take place at Φ^3.

Theorem 1. Given an extent of size s in blocks, the number of allocation matrices for the segment partitioning is $\log_2(s/\lambda)$. Also, using Φ^L, the maximum number of segments available at level *L* is:

$$\sum_{i=a}^{b}(x[i,H] + \sum_{k=0}^{i-1} x[i,k]),\ a = \log_2 \lambda \text{ and } b = (\log_2 s) - 1$$

Proof. In MatBall, the segments with indices between $\log_2 \lambda$ and (log₂s)-1 are partitioned at each level. Therefore, the number of levels for the segment partitioning is $\log_2 s - \log_2 \lambda$, resulting in $\log_2(s/\lambda)$ allocation matrices to be created. In Φ^L, the number of segments to be generated from segment *i* of level *L*-1 is $x[i,H]+\sum_{j=0}^{i-1}x[i,j]$. Also, the segments that are subject to the segment partitioning at level *L*-1 are from $\log_2\lambda$ to (log₂s)-1. Thus, the number of segments available at level *L* is $\sum_{i=\log_2\lambda}^{(\log_2 s)-1}(x[i,H]+\sum_{j=0}^{i-1}x[i,j])$

In Figure 2(b), *seg*[*,5] of the top level is partitioned to {*seg*[5,*H*], *seg*[5,0], *seg*[5,1], *seg*[5,2], *seg*[5,3],*seg*[5,4]}. Since only a single child with such an index is available at level one, the associated elements in Φ^1 is set to one. The same procedure is applied to *seg*[*,6] and *seg*[*,7]. As a result, the total number of segments available at level one is $\sum_{i=5}^{7}(x[i,H]+\sum_{j=0}^{i-1}x[i,j])=21.$

C. Segment Mapping

In this Section, we describe a hierarchical, fine-grained way of mapping data to segments consisting of extents. The segment mapping of MatBall was designed to collect data in an extent as many as possible, to improve the space utilization of extents. Also, the starting position of the segment mapping can be easily calculated by referencing the hierarchical structure of extents.

Definition 3 (Segment mapping) Given Φ^L = *x*[*N*+1,*N*], assume that a sequence of the segment partitioning for row *i* (*i*>*H*) is given by

$$seg[*,a]\rightarrow seg[a,b]\rightarrow\cdots\xrightarrow{L-1}seg[o,i]\xrightarrow{L}seg[i,j]$$

Then the starting block position of segment *j* at *L* is $pos(seg[i,j])=2^a+2^b+\cdots+2^i+2^j.$

Since the block position of the head segment is ze ro, *pos*(*seg*[*,*a*]) is 2^a. Furthermore, child *b*>*H* of level one has the size of 2^b in blocks, thus *pos*(*seg*[*a*,*b*])=2^a+2^b. As a result, segment *j* generated from *i* has the starting block position of $2^a+2^b+\cdots+2^i+2^j$. Likewise, for the children of segment *j*, *pos*(*seg*[*j*,*H*]) = *pos*(seg[*i*,*j*]) and *pos*(seg[*j*,*k*])=*pos*(*seg*[*i*,*j*])+2^k where *k*>*H.*

For example, in Figure 1, suppose that a file has been allocated to an extent from block position zero to 165. First of all, the ending block position 165 falls into segment seven. Due to its size larger than λ, the partitioning at the level one takes place:

$$\text{I)}\xrightarrow{level0}2^7\le 165<2^8,$$
$$size(seg[*,7])=2^7>\lambda,\ pos(seg[*,7])=2^7$$

The child segment five being partitioned from segment seven of the top level contains the block position 165 and the starting position of child five is calculated by adding its size to its parent starting position:

$$\text{II)}\xrightarrow{levle1}2^5\le 165-pos(seg[*,7]<2^6,$$
$$size(seg[7,5])=2^5>\lambda,\ pos(seg[7,5])=pos(seg[*,7])+2^5$$

Since the size of child five is still the same to λ, one more partitioning at level two takes place, resulting in mapping block potion 165 to segment two at level two. The starting position of segment two is obtained by applying the same way we did in the upper level:

$$\text{III)}\xrightarrow{levle2}2^2\le 165-pos(seg[7,5]<2^3,$$
$$size(seg[5,2])=2^2<\lambda,\ pos(seg[5,2])=pos(seg[7,5])+2^2$$

As a result, the next file allocation in the same extent takes place from segment three of level two that begins at block position 168:

pos(*seg*[5,3]) = *pos*(*seg*[*,7])+*pos*(*seg*[7,5])+2^3=168.

Algorithm: *MAP* (input:*w*, output:*level*, *index*, *next*)

```
1.  compute j such that $2^j \le w < 2^{j+1}$; level=0;
2.  if $j < \log_2\lambda$
3.      index=k+1; next=pos(seg[*,j+1]);
4.      return
5.  end if
6.  pos = pos(seg[*,j]);
7.  while $j \ge \log_2\lambda$ do
8.      level ++;
9.      find k such that $2^k \le w - pos < 2^{k+1}$;
10.     if $k < \log_2\lambda$
11.         index=k+1; next=pos(seg[j,k+1]);
12.         return
13.     end if
14.     pos = pos(seg[j,k]);
15.     j= k;
16. end while
```

Figure 3. File allocation algorithm on extents.

Figure 3 shows the steps involved in finding the segment where the next file allocation begins on the extent. Let *w* be the ending block position of the last file allocation. The output of the algorithm is *level*, *index* and block position *next* where the next file allocation starts. In the algorithm, step 2 to 5 executes file allocation without the segment partitioning and takes O(1). Step 7 to 16 shows the segment partitioning taking place when the size of segment mapped to *w* is larger than or equal to λ. Since the maximum number of the segment partitioning is $\log_2(s/\lambda)$, the time complexity of the algorithm is O(log$_2$*s*). In MatBall, only the extents containing at least λ free blocks are reused for space utilization.

Theorem 2. Given an extent *E* of size *s* in blocks, let $\lambda = 2^n (n \geq 1)$ and *L* be the number of levels of the segment partitioning. Then, with $\delta = 2^m (m \geq 1)$ such that $\delta < \lambda$, the levels needed for partitioning is $L + \log_2(\lambda / \delta)$. Also, let *w* ($s - w \geq \lambda + 1$) be the ending block position of the last file allocation of *E* and *p* and *q* be the hole sizes with λ and δ, respectively. Then $p \geq q$.

Proof. With δ and λ, since the number of levels for the segment partitioning is $\log_2(s/\delta)$ and $\log_2(s/\lambda)$, the level difference between two thresholds is $\log_2(\lambda/\delta)$. Therefore, for $\delta(<\lambda)$, it needs $L + \log_2(\lambda/\delta)$ partitioning levels at maximum. Assume that *w* is mapped to segment *i* at level *X* on *E*. Let *o* be the parent of *i* at level ***X***-1.

case $(i < \log_2 \delta)$: no segment partitioning takes place on *i* with two thresholds. In this case, the next allocation occurs at *pos*(*seg*[*o*,*i*+1]). As a result, *p*=*q*= *pos*(*seg*[*o*,*i*+1])-(*w*+1).

case $(\log_2 \delta \leq i < \log_2 \lambda)$: *i* is not partitioned with λ and thus *p* = *pos*(*seg*[*o*,*i*+1])-(*w*+1). On the other hand, with δ, segment *i* is partitioned into the lower level *X*+1. Let *k* be the segment of *X*+1 where *w* is mapped. Then, the next file allocation on *E* begins at segment *k*+1. Since *pos*(*seg*[*i*,*k*+1]) < *pos*(*seg*[*o*,*i*+1]), *q* = *pos*(*seg*[*i*,*k*+1])-(*w*+1) < *p*.

case $(i \geq \log_2 \lambda)$: segment *i* with δ is more partitioned than with λ due to $\delta < \lambda$. Also, the more a segment is partitioned, the smaller the hole size is between two consecutive file allocations on *E*. Therefore, *q*<*p*.

Theorem 2 implies that there is a tradeoff between partitioning overhead and space utilization, regarding to the threshold value. With the small threshold value, the hole size between two consecutive file allocations on an extent becomes small. However, it might need more partitioning steps than with a larger value. Our objective in using the allocation matrix is to choose the appropriate partitioning threshold to reduce extent fragmentation while minimizing the partitioning overhead.

III. PERFORMANCE EVALUATION

We present the performance measurement of MatBall.

A. Experimental Platform

We integrated MatBall with the hybrid file system. In the hybrid file system, the entire address space is constructed by combining a small portion of SSD partition with HDD partition. The file allocation of SSD partition is executed by performing MatBall, therefore the files are allocated per extent composed of a group of segments.

When the hybrid file system is mounted, the clean extents that are not used for file allocations yet and the allocation matrices are organized in memory. Also, I/O request is simultaneously performed on both partitions. When either of partitions completes I/O, control returns user.

Table 1 illustrates the number of allocation matrices and partitioning description for each extent size and threshold value λ. We evaluated MatBall with IOzone while comparing it with ext2 installed on HDD and SSD.

TABLE Ⅰ SEGMENT PARTITIONING BASED ON EXTENT SIZE

extent size	λ	# of allocation matrices	partitioning description
64	16	2	$\xrightarrow{level0} \{seg[*,H], seg[*,0], \cdots, seg[*,5]\}$ *seg*[*,4] and *seg*[*,5] are involved in the subsequent segment partitioning.
	32	1	$\xrightarrow{level0} \{seg[*,H], seg[*,0], \cdots, seg[*,5]\}$ only *seg*[*,5] is involved in the subsequent segment partitioning.
256	16	4	$\xrightarrow{level0} \{seg[*,H], seg[*,0], \cdots, seg[*,7]\}$ *seg*[*,4] to *seg*[*,7] are involved in the subsequent segment partitioning.
	32	3	$\xrightarrow{level0} \{seg[*,H], seg[*,0], \cdots, seg[*,7]\}$ *seg*[*,5] to *seg*[*,7] are involved in the subsequent segment partitioning.

The performance measurements are executed on a PC with AMD Athlon dual-core processor and 1GB of memory. The HDD partition is equipped with a 320GB of Seagate 7200 RPM disk and SSD partition uses fusion-io SSD ioDrive. We used CentOS release 6.2 with a 2.6.18 kernel. The hybrid file system integrated with MatBall uses database built on SSD partition.

B. IOzone Experiment

We evaluated MatBall using IOzone, while varying file sizes from 64KB to 256MB. We executed *./iozone –azi –e –g* 256M *–q* 8K to use 8KB of I/O record size. We used *fsync*() in every I/O operations to reduce the effect of memory cache. Also, for each I/O operation, we changed the threshold value for the segment partitioning from 16 to 32, to observe the partitioning overhead.

Figure 4 and 5 show the write performance of MatBall, while comparing it to that of ext2 installed on HDD and SSD. The extent sizes of MatBall are 64KB and 256KB and $\lambda = 16$. As can be seen in the figure, the write performance of ext2 on HDD is much lower than that of MatBall because of the hybrid structure of MatBall.

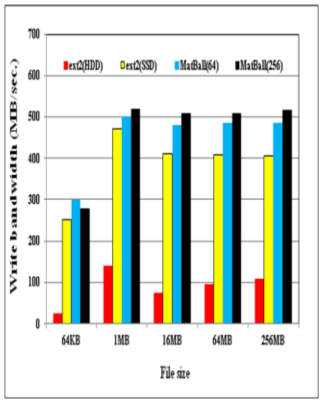

Figure 4. Write $\lambda = 16$

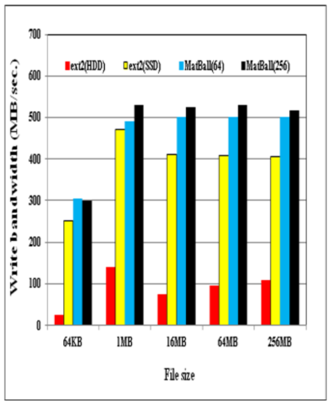

Figure 5. Write $\lambda = 32$

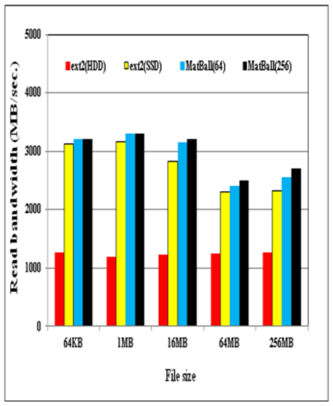

Figure 6. Read $\lambda = 16$

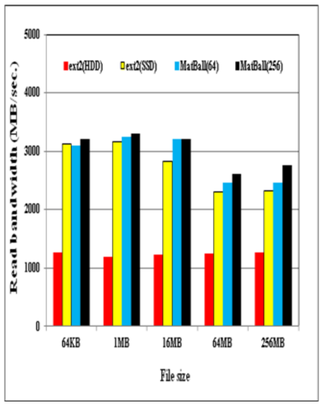

Figure 7. Read $\lambda = 32$

When the write throughput of MatBall composed of 64KB of extents is compared to that of ext2 installed on SSD, with 256MB of file size, there is about 19% of performance improvement. This is because MatBall uses the large I/O units for write operations.

The advantage of using the large I/O granularity is also observed in the write performance of MatBall composed of 256KB of extent sizes. The figure shows that MatBall using 256KB of extent sizes produces about 6% performance improvement compared to MatBall with 64KB of extents.

However, in write operations on small files such as 64KB of files, using the larger extent size does not produce the performance speedup. According to Figure 4, on top of 64KB files, using 256KB of extent size rather decreases the write performance, when compared to using 64KB of extent size. This is because of the overhead for coalescing data on extents. In other words, with small files, the larger the extent size is the more the overhead for collecting data on extents takes place. However, on top of large files, writing with the large I/O granularity increases the write throughput due to the reduction in I/O accesses.

Figure 5 shows the write performance with $\lambda = 32$. Since the file sizes of IOzone are aligned with extent sizes, no remaining blocks on extents are left posterior to file allocations. As a result, only the segment partitioning on the top level takes place for both extent sizes.

For example, with 64KB of extent sizes, only seven segments on the top level are configured and used for file allocations. Every file size uses the entire 64KB of extents although multiple extents are used for files larger than 64KB. Therefore, no segment partitioning is needed to reuse extents. The same I/O behavior can be observed with $\lambda = 16$ in Figure 4. As a result, there is no noticeable difference between the write performances with $\lambda = 32$ and that of $\lambda = 16$.

Figure 6 and 7 show the read performance with $\lambda = 16$ and with $\lambda = 32$, respectively. In this case, we can see that the memory cache significantly affects in the read performance. Figure 6 shows that the prefetching scheme implemented in ext2 offsets the performance difference caused by device characteristics a little. Because the difference of the read performance between ext2 on HDD and ext2 on SSD is lower than that of the write difference of ext2 between two devices.

Likewise, we cannot find the noticeable difference between MatBall with 64KB of extent sizes and that with 256KB of extent sizes. However, because of the less read accesses, on top of 256MB of file sizes, using 256KB of extent sizes produces 5% of performance speedup over 64KB of extent sizes. Also, in Figure 6 and 7, we can see that changing the threshold value for the segment partitioning does not effect on the read performance because read operations are not involved in the segment partitioning.

IV. CONCLUSION

The main goal of MatBall is to increase the space utilization of SSD partition in the hybrid file system where the entire address space is provided by integrating a small portion of SSD partition with a much larger HDD storage capacity. MatBall tries to consume the remaining spaces as much as possible posterior to file allocation processes, by recursively partitioning segments in the subsequent level and by allowing the further file allocations on the partitioned segments. We evaluated MatBall using IOzone. When file sizes are either a multiple of extent sizes or larger than the extent size, the segment partitioning to the lower level does rarely take place. In this case, the threshold value for the segment partitioning in MatBall does little affect I/O performance. On the other hand, with a large number of small-size files, MatBall can improve I/O bandwidth by converting data into the larger I/O granularity. As a future work, we will verify the effectiveness and suitability of file allocation method of MatBall by using various applications.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP)(NRF-2014R1A2A2A01002614).

Also, this work was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIP) (No.B0101-15-0548(2016), Development of Ultra Power-Saving and High Integrated Micro Server Technology for Cloud Infra).

REFERENCES

[1] N. Agrawal, et al., "Design Tradeoffs for SSD Performance," In Proceedings of USENIX Annual Technical Conference, pp. 57-90, June 2008.

[2] M. Saxena, M. Swift, and Y. Zhang, "FlashTier: a Lightweight, Consistent and Durable Storage Cache," In Proceedings of EuroSys'12, pp. 267-280, 2012.

[3] 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, Vol. 29, pp. 953-965, 2010.

[4] A. Rajimwale, V. Prabhakaran, and J.D. Davis, "Block Management in Solid-State Devices," 2009 USENIX Annual Technical Conference, pp. 21-26, June 2009.

[5] H. Kim, S. Seshadri, C. Dickey, and L. Chiu, "Evaluating Phase Change Memory for Enterprise storage Systems: A Study of Caching and Tiering Approaches," In Proceedings of the 12th USENIX conference on File and Storage Technologies, Santa Clara, USA, pp. 33-45, 2014.

[6] W. Josephson, L. Bongo, K. Li, and D. Flynn, "DFS: A File System for Virtualized Flash Storage," ACM Transactions on Storage, Vol. 6, No. 14, pp.1-15, Sept. 2010.

[7] C, Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," In Proceedings of the 13th USENIX conference on File and Storage Technologies, Santa Clara, USA, pp. 273-286, 2015.

[8] J. Kang, et al.,"SpanFS: A Scalable File System on Fast Storage Devices," In Proceedings of USENIX Annual Technical Conference, Santo Clara, USA, pp. 249-261, 2015.

[9] C. Li, et al., "Nitro: A Capacity-Optimized SSD Cache for Primary Storage," In Proceedings of USENIX ATC'14, Philadelphia, USA, pp. 501-512, 2014.

[10] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "FlexECC: Partially Relaxing ECC of MLC SSD for better cache performance," In Proceedings of USENIX Annual Technical Conference, Philadelphia, USA, pp. 489-500, 2014.

[11] Z. Zhang and K. Ghose, "hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance," EuroSys'07 , pp. 175-187, 2007.

[12] E.Gal and S. Toledo, "A Transactional Flash File System for Microcontrollers", In Proceedings of the USENIX Annual Technical Conference, pp. 89-104, April 2005.