Proc. of The Third Intl. Conf. on Advances in Computing, Electronics and Communication - ACEC 2015
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-064-4 doi: 10.15224/ 978-1-63248-064-4-28

Exploiting Self-Adaptive, 2-Way Hybrid File Allocation Algorithm

[Jaechun No, Sung-Soon Park]

Abstract— **We present hybridFS file system that provides a hybrid structure, which makes use of performance potentials of NAND flash-based SSD (Solid-State Device). As the technology of flash memory has rapidly improved, SSD is being used in various IT products as a nonvolatile storage media. Applications looking for better I/O performance attempt to achieve desirable bandwidth, by employing SSD to storage subsystems. However, building a large-scale SSD storage subsystem deploys several issues that need to be addressed. Those issues include peculiar physical characteristics related to flash memory and high SSD cost per capacity compared to HDD devices. This paper presents a new form of self-adaptive, hybrid file system, called hybridFS, which properly attempts to address aforementioned issues. The main goal of hybridFS is to combine attractive features of both HDD and SSD devices, to construct a large-scale, virtualized address space in a cost-effective way. The performance evaluation shows that hybridFS generates a comparable bandwidth to that of file system installed on SSD devices, while offering a much larger storage capacity.**

Keywords—SSD, transparent file mapping, data section, 2-way extent allocation, data layout

I. Introduction

In this paper, we present hybridFS file system whose main goal is to integrate the advantages of both SSD and HDD devices in a cost-effective way. As the potentials of SSDs have been recognized, such as high random I/O performance and low-power consumption[1,13], SSD is widely being used in IT products these days. The performance superiority of SSDs over HDDs becomes a driving force of numerous researches related to SSD, with the expectation of generating high I/O performance in various applications. For instance, multimedia or database applications can obtain performance benefits by employing SSD storage subsystems, to serve a large number of I/O requests.

Jaechun No
College of Electronics and Information Engineering
Sejong University, Korea

Song-Soon Park
Dept. of Computer Engineering
Anyang University and Gluesys Co. LTD, Korea

Despite its performance potentials, the common usage of SSD is currently restricted to small-size memory devices, such as mobile equipments. The key obstacle to the widening SSD adoption to large-scale storage subsystems is its high cost per capacity ($3/GB for SSD, whereas $0.3/GB for HDD)[1]. Even though the cost of flash memory becomes decrease, the price of SSD is still much higher, compared to that of HDD. Such a high cost/capacity ratio makes it less desirable to build large-scale storage subsystems solely composed of SSD devices. An alternative storage solution is to build a hybrid storage subsystem where both SSD and HDD devices are combined in a cost-effective way, while making use of the strengths of both devices.

In this paper, we present a hybrid file system, called hybridFS, developed for exploiting the hybrid storage structure. HybridFS is capable of generating comparable performance to the file system installed on SSD devices, while offering much larger storage capacity at less cost. This is achieved by integrating vast, low-cost HDD storage space with a small portion of SSD space through several optimization schemes, to address the strengths and the shortcomings of both devices. In this paper, we will discuss how we implemented hybridFS to make use of SSD's high I/O throughput, while providing a flexible internal structure to retain high sequential read performance of existing file systems on HDD devices.

This paper is organized as follows: In Section 2, we discuss the design motivations and related studies. In Section 3, we describe the detailed implementation issues of hybridFS. In Section 4, we present the performance measurements of hybridFS and in Section 5, we conclude with a summary.

II. Related Study

As pointed out by [2,3], the major drawbacks of flash memory are write latency due to erasure per block and cleaning problems. Many flash file systems [4,5,6] take the out-of-place approach proposed by log-structured file system [7], to reduce the semiconductor overhead of flash memory.

The well-known characteristic of log-structured file system is its sequential, out-of-place update using logs. The logs are divided into segments, to maintain large disk free area and to avoid space fragmentation. The garbage collection is performed per segment, by copying live data out of a segment. This update approach has been adopted to most flash file systems, to minimize the block erasure overhead.

JFFS and JFFS2 [6] maintain file metadata logs for the sequential writing. To mount file system, they require to scan all the logs in flash memory, which may take considerably long time to build. Also, both flash file systems

SEEK DIGITAL LIBRARY

Proc. of The Third Intl. Conf. on Advances in Computing, Electronics and Communication - ACEC 2015
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-064-4 doi: 10.15224/ 978-1-63248-064-4-28

were designed for embedded equipments with small-size NAND flash
memory.

TFFS [5] is an optimized file system which suits for the small embedded systems with less than 4KB of RAM. TFFS is targeted for NOR devices and provides several useful functions, such as tailored API for the embedded device and concurrent transaction recoveries. ELF [4] is built for sensor nodes and keeps a log entry for each flash page. ELF tries to optimize the logging overhead by reducing the number of log entries. However, most of flash file systems have been developed for small-size flash memory and therefore is not appropriate for a large-scale data storage.

As hybridFS does with SSD devices, there are several interesting attempts to make use of storage class memories. LiFS [8] intends to store file system metadata to smaller, faster storage class memory, such as MRAM. Conquest [9] uses persistent RAM for storing small files and file system metadata. Only the remaining data of large files are stored in disk. MRAMFS [10] goes further by adding compression, to overcome the limited size of NVRAM. HeRMES [11] proposed to keep all file metadata in MRAM and to apply compression to minimize the required space for metadata. hFS [12] attempts to combine the strengths of both FFS and log-structured file system, by separating file data and file metadata into two partitions.

Besides, Agrawal et. al.[13] describes interesting SSD design issues using Samsung's NAND-flash. They also suggest several ways of obtaining improved I/O performance on SSD, by analyzing a variety of I/O traces extracted from real systems. However, there is little file system works for hybrid storage subsystems where HDD and SSD are incorporated to provide a large-scale, virtualized address space.

III. Hybrid File Mapping

A. *Logical Data Layout*

HybridFS was developed to exploit SSD performance advantages and data layout flexibility. The entire address space of hybridFS is constructed by combining two physical partitions: SSD partition and HDD partition. The SSD partition of hybridFS works as a write-through persistent cache and stores recently referenced files recognized by file access time. Also, it stores the metadata being used for allocating SSD extents and logs. On the other hand, HDD partition of hybridFS stores the backup-ed file data and most of file system metadata.

The SSD partition of hybridFS was designed to retain SSD's performance potentials and to provide the transparent file mapping to SSD data layout. To generate improved I/O performance, hybridFS attempts to reduce allocation overhead, by providing the flexible data layout and the extent alignment to flash block size on file system level. The layout flexibility is accomplished by defining multiple data sections with the different extent size. Also, aligning extent size to the size of flash block is performed on file system level because hybridFS is not allowed to access physical flash block through FTL embedded in SSD.

The transparency of file mapping is performed by separating the logical directory hierarchy from the file mapping to SSD data section. The file mapping to SSD data section is initialized at file system creation by reading the configuration file and is stored in the in-memory map table. In hybridFS, a file can be mapped to the appropriate data section, according to file size, usage, and access pattern. Furthermore, a file which does not need fast I/O processing speed, such as snapshot images, can bypass the caching to SSD partition. HybridFS also enables to move sub-level directories and their files to another data section without modifying the hierarchical structure, in case that the file access pattern or usage is changed.

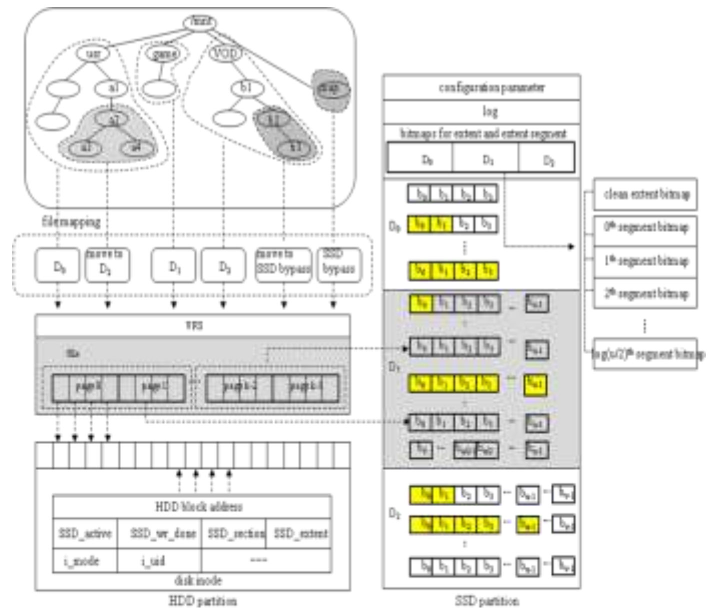

Figure 1. An overall structure of hybridFS

Figure 1 illustrates an overview of hybridFS data layout. As can be seen in the Figure, I/O unit of SSD partition is an extent, whereas I/O unit of HDD partition is a block. Also, in the Figure, three logical data sections, D_0, D_1, and D_2, are defined with the different extent size, that is *4* for D_0, *u* for D_1, *v* for D_2 respectively, where $4<u<v$. The information about SSD configuration, including the number of data sections, section range, and extent size of each data section, is determined at file system creation.

Such a layout configuration enables to maintain SSD address space, according to file characteristics. For example, files with a large, contiguous access pattern, such as video files, can be assigned to the data section composed of a large extent size. This layout flexibility is also reflected to the file mapping. For instance, in Figure 1, since most files in /mnt/VOD have a large, contiguous access pattern, D_2 with the largest extent size is mapped to the directory. Also, the files stored in /mnt/snap are configured to bypass SSD cache, to prevent the costly SSD space from being consumed by unnecessary files. Figure 1 also shows that the mapping to the data section can be changed to another data section without impacting the hierarchical structure. For example, */mnt/usr/a1/a2* is remapped from D_0 to D_2 and the mapping for */mnt/VOD/b1/b2* is changed from D_2 to SSD bypass.

B. *File Allocation*

In hybridFS, the file allocation on SSD partition is executed on per extent. In designing the file allocation

SEEK DIGITAL LIBRARY

Proc. of The Third Intl. Conf. on Advances in Computing, Electronics and Communication - ACEC 2015
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-064-4 doi: 10.15224/ 978-1-63248-064-4-28

scheme, the key objectives were to classify SSD allocation space according to file access pattern and usage, to efficiently manage the costly storage capacity, and to align the data size to flash block size to minimize the erasure latency. Classifying SSD allocation space is performed by defining the different extent size for each data section and then by mapping files to the data section, in terms of their access characteristics. Also, hybridFS attempts to align file allocation to the boundary of flash block, to control the block erasure overhead on the file system level.

However, the shortcoming of hybridFS extent scheme is the extent fragmentation where a portion of extent is left unused. Furthermore, constantly aligning extent size to flash block may be difficult, especially in such a case that the extent size is smaller than the size of flash block.

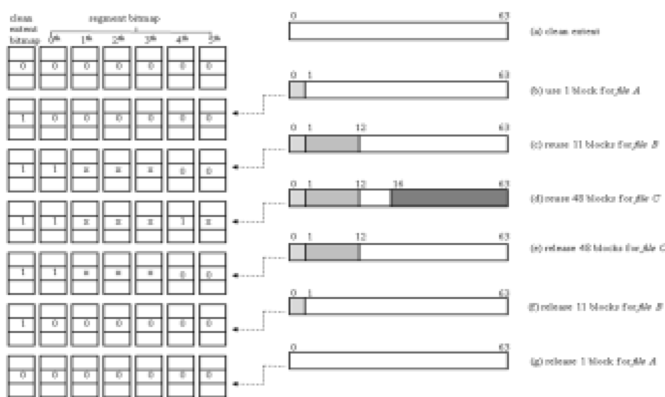
Figure 2. File allocation procedure

In hybridFS, the file allocation on SSD partition is executed on per extent. In designing the file allocation scheme, the key objectives were to classify SSD allocation space according to file access pattern and usage, to efficiently manage the costly storage capacity, and to align the data size to flash block size to minimize the erasure latency. Classifying SSD allocation space is performed by defining the different extent size for each data section and then by mapping files to the data section, in terms of their access characteristics. Also, hybridFS attempts to align file allocation to the boundary of flash block, to control the block erasure overhead on the file system level.

However, the shortcoming of hybridFS extent scheme is the extent fragmentation where a portion of extent is left unused. Furthermore, constantly aligning extent size to flash block may be difficult, especially in such a case that the extent size is smaller than the size of flash block.

To alleviate the extent fragmentation problem, hybridFS divides the extents into two groups: one group for the clean extent where all the blocks are free and the other group for the extent segment where a portion of blocks in an extent are unused. Using the extent segment might sacrifice the performance advantage of SSD because the file allocation on the extent segment might not be aligned to the boundary of flash block. However, we decided to concentrate on making use of the costly SSD address space as much as possible.

With an extent composed of x number of blocks, there exist $log(x)+1$ number of extent bitmaps, one for the first group and $log(x)$ number of segment bitmaps for the second group. HybridFS reuses only extent segments whose remaining number of free blocks is more than or equal to half of total blocks in an extent, to prevent widespread file allocation across extent segments. The first bitmap of the second group, that is 0th segment bitmap, shows the allocation status of the extent segments where $x-1$ number of blocks is unused. Similarly, i^{th} segment bitmap indicates whether the extent segments with $x-2^i$ $(0 \leq i \leq log(x)-1)$ number of blocks are used.

Figure 2 illustrates the file allocation process with the extent consisting of 64 blocks. This procedure is the default allocation process for SSD partition and is applied in such a case that either the extent size of a data section is equal to or larger than the flash block size, or the SSD flash block size is not clearly specified. In Figure 2, there exist seven extent bitmaps, one for the clean extent and the other six for the extent segments. Searching for an extent is started by scanning the clean extent bitmap.

Suppose that *file A* needing 1block is assigned to the clean extent, as shown in Figure 2(b). In this case, the bit of the clean extent bitmap is set to one. In Figure 2(c), if *file B* requires 11 blocks to allocate, hybridFS then reuses the same extent to store data, while setting the associated bit of 0th segment bitmap to one. The corresponding bits from 1th to 3th segment bitmaps would be unavailable because, after the file allocation of *B*, the remaining number of unused blocks is smaller than that indicated by those bitmaps.

In Figure 2(d), if *file C* needs 48 blocks to store new data, then the associated bit of 4th segment bitmap is set to one, to indicate the corresponding extent no longer to be a candidate for the file allocation. Upon releasing all the files, the extent returns to the clean state.

When the extent size of the data section is smaller than the flash block size, hybridFS defines a virtual extent composed of multiple, real extents. The size of a virtual extent matches the size of flash block and the allocation policy including extent bitmap and extent segment is executed on per virtual extent. With a virtual extent consisting of y number of extents, there exists $log(y)$ $+1$ number of virtual extent bitmaps, one for the virtual clean extent and the other $log(y)$ number of bitmaps for the virtual extent segment. The virtual extent bitmaps indicate the allocation status of virtual extents, in a similar way as hybridFS did with real extents composed of x number of blocks. In other words, i^{th} virtual segment bitmap indicates whether the virtual extent segments with $y-2^i$ $(0 \leq i \leq log(y)-1)$ number of extents are used.

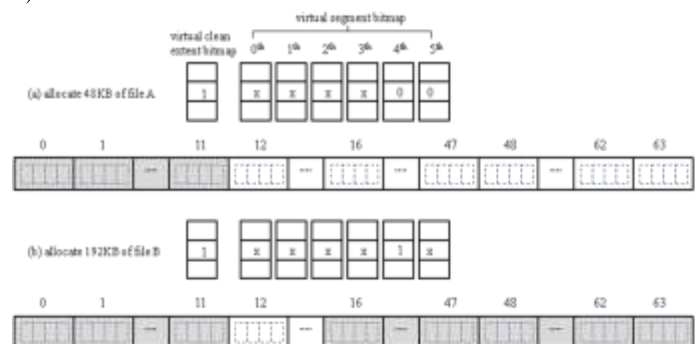
Figure 3. Virtual extent allocation

Figure 3 shows how the virtual extent segment is organized in the data section composed of 4KB of extents,

SEEK DIGITAL LIBRARY

Proc. of The Third Intl. Conf. on Advances in Computing, Electronics and Communication - ACEC 2015
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-064-4 doi: 10.15224/ 978-1-63248-064-4-28

on top of SSD partition with 256KB of flash block. In this case, a virtual extent is consisted of 64 4KB of real extents and the seven virtual extent bitmaps are created to indicate the allocation status of virtual extents. In Figure 3, when file A stores 48KB of data, the bit of the virtual clean extent bitmap is set to one, followed by setting the associated bits from 0th to 3th virtual segment bitmaps to be unavailable. On the 192KB of allocation request, hybridFS reuses the virtual extent segment starting from the 16th real extent, while setting the associated bit of 4th virtual segment bitmap to one.

IV. Performance Evaluation

We evaluated I/O performance of hybridFS, comparing to the performance of two file systems: ext2[14] and xfs[15]. We chose ext2 for the performance comparison because the block allocation implemented on HDD partition is similar to ext2. The comparison with ext2 gives an opportunity to observe how significantly SSD partition of hybridFS affects the performance of various file operations.

The reason for choosing xfs for the performance comparison is its extent-based allocation using B+ tree. HybridFS uses the pre-determined extent allocation to be configured at file system creation. We will observe how differently both allocation schemes work on several file operations.

The experimental platform has Intel Xeon 3GHz CPU, 16GB of RAM, 750GB of SATA HDD, and 80GB of fusion-io SSD. We compared I/O performance of three file systems, using IOzone benchmark[16], on top of HDD and SSD devices. In IOzone benchmark, in case of read evaluations, we saw that the impact of memory cache greatly contributes to generate high I/O bandwidth.

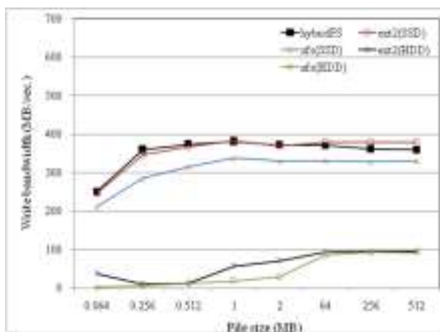

Figure 4. IOzone write

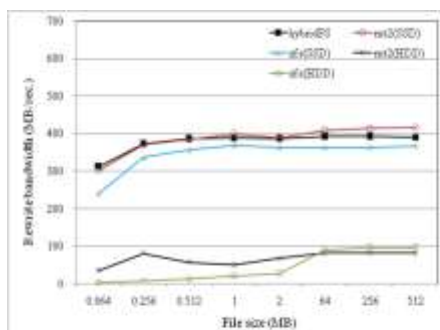

Figure 5. IOzone rewrite

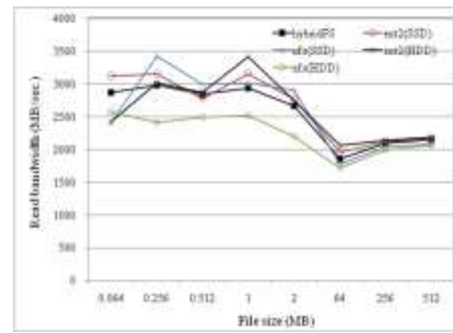

Figure 6. IOzone read

Figure 4 shows the write bandwidth on both devices. As can be seen, the performance of hybridFS is almost three times higher than that of both ext2 and xfs installed on HDD devices, whereas is 12% faster than xfs on SSD device.

Even though the performances of ext2 and xfs on SSD devices are much higher than those on HDD devices, it is less desirable to build a large-scale storage subsystem solely composed of SSD devices, due to the limited storage capacity and high cost. The performance shows that in such a case, hybridFS offers an alternative way of utilizing SSD’s high performance and HDD’s vast storage capacity with low cost.

The performance evaluation of write operations on two devices points out several interesting aspects. First, the difference between the worst and best performance of ext2 and xfs on HDD devices is much larger than the corresponding difference on SSD devices. For example, on top of HDD device, when the performance with 512MB of file size is compared to that with 256KB of file size, ext2 shows almost eight times speed improvement. On the other hand, on top of SSD device, the performance difference between 64KB and 512MB of file sizes is about 35%, which is much less than that on HDD device.

Xfs also shows a similar behaviour to ext2, resulting in about 33% of performance speedup between 64KB and 512MB of file sizes on SSD device, while producing much higher difference between the same file sizes on HDD device. This indicates that the overhead of HDD moving parts more deteriorates write performance than the overhead of SSD semiconductor property does. Due to the absence of moving overhead, both ext2 and xfs on SSD devices generate almost three times higher bandwidth than those on HDD devices.

Second, although hybridFS uses HDD portion as metadata store, such an internal structure does not greatly degrade write performance because we can observe that, except for 64KB of file size, the performance difference between small-size and large-size files is very small even though more metadata accesses occur in the small-size files than in the large-size files.

In Figure 5, we compared the rewrite performance of hybridFS to that of both ext2 and xfs. Similar to the write performance on HDD device, the rewrite throughput of hybridFS outperforms that of both ext2 and xfs installed on HDD devices. When compared to both file systems on SSD devices, hybridFS generates almost the same bandwidth with ext2, but is marginally faster than xfs.

In the file read operation (Figure 6), unlike in the write experiments on HDD device, hybridFS does not produce

SEEK DIGITAL LIBRARY

Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-064-4 doi: 10.15224/ 978-1-63248-064-4-28

noticeable performance difference. For ext2 and xfs, the performance difference between worst and best cases on both devices is much less than that in the write experiment. This is because, in the IOzone benchmark, most read requests are served from memory cache.

V. Conclusion

As the technology of flash memory rapidly grows, SSD has drawn a great attention from IT enterprises as an attractive storage solution for fast I/O processing needs. SSD not only generates high I/O performance because of the absence of mechanical moving overhead but also provides significant power savings. However, despite its promising potentials, most SSD usages in real products have been limited to small-size memory devices, such as mobile equipments, because of its high cost per capacity. In this paper, we proposed a way of integrating SSD devices with HDD devices in a cost-effective manner, to build a large-scale, virtual address space. To achieve better I/O performance, hybridFS uses SSD partition as a write-through cache, which contains hot files recognized by file access time. Besides making use of the advantages of SSD, hybridFS attempts to provide a flexible internal structure to retain the excellent sequential read performance of existing file systems. HybridFS evaluation shows that achieving high I/O performance by combining the advantages of both SSD and HDD devices is possible. The strength of hybridFS is most noticeable when its write performance is compared to the corresponding performance of both ext2 and xfs installed on HDD devices. The write experiment indicates that the mechanical moving overhead of HDD more affects the write performance than the semiconductor overhead of SSD does. HybridFS frequently produced comparable performance to that of both ext2 and xfs installed on SSD devices. Since building a large-scale storage subsystem using SSD devices is less desirable due to high cost and limited storage resource, hybridFS can be a good alternative to make use of high I/O performance of SSD and vast storage capacity of HDD. As a future work, we would verify the performance of hybridFS by using more benchmarks.

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 Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.B0101-15-0548(2015), Development of Integrated Management Technology for Micro Server Resources).

References

[1] M. Saxena and M. Swift, "FlashVM: Virtual Memory Management on Flash," 2010 USENIX Annual Technical Conference , Boston, MA, 2010.

[2] W. K. Josephson, L. A. Bongo, and D. Flynn, "DFS: A File System for Virtualized Flash Storage," In Proceedings of the 8th USENIX Conference on File and Storage Technologies, San Jose, USA, 2010.

[3] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," In Proceedings of the 8th USENIX Conference on File and Storage Technologies, San Jose, USA, 2010.

[4] H. Dai, M. Neufeld, and R. Han, "ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes," SenSys'04, Baltimore, USA, 2004.

[5] E. Gal and S. Toledo, "A Transactional Flash File System for Microcontrollers," In Proceedings of 2005 USENIX Annual Technical Conference, Anaheim, CA, 2005.

[6] D. Woodhouse, "JFFS: The Journaling Flash File System," *In Ottawa Linux Symposium*, 2001.

[7] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log Structured File System," In Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1991, pp.1-15.

[8] S. Ames, N. Bobb, K. Greenan, O. Hofmann, M. W. Storer, C. Maltzahn, E. L. Miller, and S. A. Brandt, "LiFS: An Attribute-Rich File System for Storage Class Memories," In Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies ," College Park, USA, 2006.

[9] A. Wang, G. Kuenning, P. Reiher, and G. Popek, "Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System," In Proceedings of the 2002 USENIX Annual Technical Conference, Monterey, CA, 2002.

[10] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "MRAMFS: A compressing file system for non-volatile RAM," In Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Volendam, Netherlands, 2004.

[11] E. L. Miller, S. A. Brandt, and D. D. E. Long, "HeRMES: High-performance reliable MRAM-enabled storage," In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems, Schloss, Germany, 2001, pp.83-87.

[12] Z. Zhang and K. Ghose, "hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance," EuroSys'07, Lisbon, Portugal, 2007.

[13] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," 2008 USENIX Annual Technical Conference , 2008.

[14] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," In Proceedings of the First Dutch International Symposium on Linux, 1995.

[15] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Tech, "Scalability in the XFS File System," In Proceedings of the USENIX 1996 Technical Conference, San Diego, USA, 1996.

[16] IOzone, Available at: http://www.iozone.org

SEEK DIGITAL LIBRARY