International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:7, 2015

# Developing NAND Flash-Memory SSD-Based File System Design

Jaechun No

***Abstract*****—**This paper focuses on I/O optimizations of N-hybrid (New-Form of hybrid), which provides a hybrid file system space constructed on SSD and HDD. Although the promising potentials of SSD, such as the absence of mechanical moving overhead and high random I/O throughput, have drawn a lot of attentions from IT enterprises, its high ratio of cost/capacity makes it less desirable to build a large-scale data storage subsystem composed of only SSDs. In this paper, we present N-hybrid that attempts to integrate the strengths of SSD and HDD, to offer a single, large hybrid file system space. Several experiments were conducted to verify the performance of N-hybrid.

***Keywords*****—**SSD, data section, I/O optimizations.

## I. Introduction

As the advantages of SSD have been recognized, such as high I/O performance, reliability and low-power usage, adopting SSD to IT products is rapidly increasing, from mobile electronics to high-end storage subsystems [1]-[4].

The most attractive feature of SSD is that SSD does not generate mechanical overhead, such as seek time of HDD, in accessing data due to its flash memory components. 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.

However, there are several serious constraints in building storage subsystems solely composed of SSDs. One of such constraints is that, because of SSD's internal flash memory components, developing a file system for the SSD storage subsystem evokes some issues that have never occurred in the HDD storage subsystem, such as erasure behavior [5], [6] and wear-leveling [10], [11].

The second constraint is SSD's high cost per unit capacity, compared to HDD. Although the first constraint could be solved by using FTL [7]-[9], high SSD cost still makes it difficult to build large-scale storage subsystems using only SSD devices. An alternative is to build a hybrid storage subsystem where both HDD and SSD devices are incorporated in an economic manner, while utilizing the strengths of both devices to the maximum extent possible.

In this paper, we introduce N-hybrid (New-Form of hybrid) file system, which is capable of generating comparable performance with the storage subsystem solely composed of SSD devices, by combining vast, low-cost HDD storage space with a small portion of SSD space. This is achieved by taking advantages of SSD's high I/O performance, while providing a flexible internal structure, to integrate excellent sequential performance of existing file systems on HDD devices.

The rest of paper is organized as follows: In Section II, we describe the background works for SSD and in Section III, we present the implementation details of N-hybrid. The performance results of N-hybrid are shows in Section IV. In Section V, we conclude our paper.

## II. Background Works

On top of ext2, we conducted two kinds of experiments to obtain intuitive knowledge of I/O behavior on both HDD and SSD devices. In this experiment, we used Bonnie++ benchmark for measuring the bandwidth ratio of sequential file operations between two devices and also used our I/O template for measuring the bandwidth ratio of randomized file operations.

All the ratio values were obtained by dividing SSD bandwidth of each file size into the corresponding HDD bandwidth. Our template performs file operations, creating 10,000 random files. The following code segment shows how a file is randomly created in our template:

```
while (number_of_files_to_be_tested) {
  choose random numbers for a, b, c, srv, m;
  trail_uid++;
  sprintf(path,"/mnt/dev_mount/disk%d/test/%c/%c/
     %c/%c/%c-%d-%c@paran.com/mbox%d.dat", srv, a, b, c, a, b,
     trail_uid, c, m);
  perform I/O operations using a file named path;
}
```

The experimental platform has Intel Xeon 3GHz CPU, 16GB of RAM, 750GB of SATA HDD and 80GB of fusion-io SSD. Figs. 1 and 2 demonstrate sequential and random performance ratios between two devices. There are several interesting points worthy of observation.

In case of sequential and random write operations, there exists large performance difference between HDD and SSD devices, with less than or equal to 4KB of file size. This difference demonstrates that, with small-size files, the overhead of HDD moving parts is more significant than that of SSD semiconductor properties.

Sequential read performance differs little between two devices, irrespective of file sizes. On the contrary, the performance ratio of random reads between two devices is higher than that of sequential reads. This is because HDD's random seek significantly deteriorates random read performance.

Jaechun No is with college of Electronics and Information Engineering, Sejong University, Seoul, Rep. of Korea (phone: +82-2-3408-3747; fax: +82-2-3408-4321; e-mail: jano@ sejong.ac.kr).

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:7, 2015

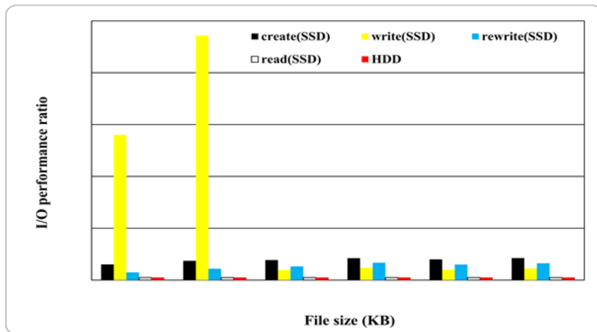


Fig. 1 Sequential comparison as a function of file size

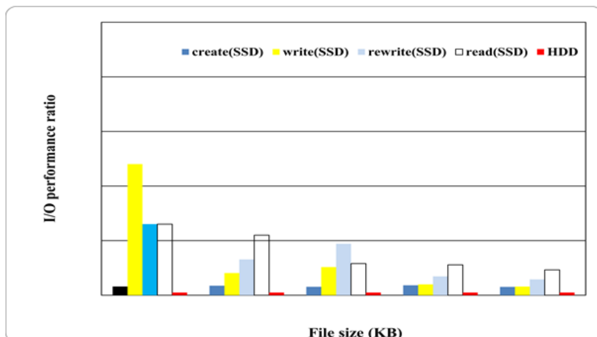


Fig. 2 Random comparison as a function of file size

Based on the results, we can assume that, for the applications where considerable I/O operations are sequential reads, HDD can generate comparable I/O performance to SSD. However, with applications generating large number of random I/O operations, using SSD as a file cache can effectively serve to speed improvement.

On top of SSD device, the performance ratio of random and sequential rewrite operations becomes comparable with more than or equal to 256KB of file size. However, with less than 256KB of file size, random rewrite operations present higher performance ratio than sequential rewrite operations. This means that as file size increases, file access pattern affects little to performance ratio between two devices. In general, the rewrite performance on SSD device is higher than that on HDD device, irrespective of file access patterns.

File creation on SSD device is effected by file access patterns, meaning that creating sequential files is faster than creating random files. We believe that ext2 strategy for creating metadata might alleviate SSD overhead in the sequential file allocation.

## III. Implementation Details

### *A. Disk Layout*

The primary objective of N-hybrid is to take potential benefits of HDD and SSD storage mediums. To utilize SSD's high-speed I/O bandwidth, N-hybrid uses SSD as a write-through persistent cache. The HDD partition of N-hybrid uses a similar allocation method to ext2/3.

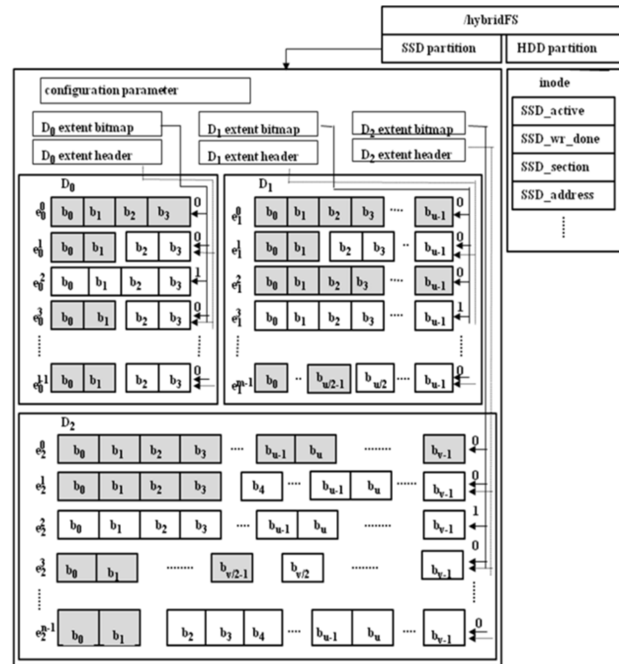


Fig. 3 Disk layout of N-hybrid: The shadowed box indicates the partially occupied data

The data structure of the file system to be built on top of HDD partition needs not significantly be modified to use SSD cache; thereby most existing kernel modules can be used without major changes to integrate SSD cache. We believe that such an N-hybrid internal structure contributes to improve portability, by minimizing manpower to switch the file system of HDD partition to another.

The N-hybrid takes different block allocation policy for both partitions. The efficiency of SSD cache depends on how effectively SSD's limited space capacity can be managed. This involves the consideration for the allocation cost of new files and for the hit ratio of most demanding files. In order to alleviate the allocation cost, N-hybrid eliminated indirectness in SSD block allocation. In other words, there is no indirect block need to be addressed. Also, to allocate SSD blocks as sequentially as possible, file creation is performed on per-extent.

Finally, N-hybrid allows defining multiple data sections in which the extent size of each section differs from each other. In N-hybrid, allocating new files can be done in two ways. First, with selective file mapping enabled, new files belonging to a directory are allocated in the data section that has been mapped to the directory at file system creation time. If no selective mapping is specified, a new file is then allocated in the data section whose extent size is most suitable for the file.

Fig. 3 illustrates the disk layout of N-hybrid. The file allocation in HDD partition is performed by using ext2/3 allocation modules. On the other hand, allocating files in SSD partition follows the different process. As shown in Fig. 3, SSD partition is divided into multiple data sections, with each section defining its own extent size. The first data section, $D_0$,

uses the extent composed of four blocks, whereas each extent of the second and third data sections, $D_1$ and $D_2$, is consisted of $u$ and $v$ blocks, respectively, where $u > 4$ and $v > u$. Providing multiple extent sizes gives an opportunity for space optimization, such as allocating large files in the data section with large extent size and allocating small files in the data section with small extent size.

The beginning of SSD partition contains the information about data sections, such as the number of data sections, section size, and pre-determined extent size of each data section. These configuration parameters are defined at file system creation. The default value for the number of data sections is one, and in such a case, an extent is composed of four blocks.

The configuration parameters are immediately followed by the extent bitmaps and headers. Each data section maintains its own bitmap and header. Each bit of the extent bitmap indicates the allocation status of the associated extent. The bit is set to one only when the whole blocks of an extent are free. The allocation status of the partially occupied extents (partial extents) is indicated by the extent header.

Fig. 4 shows the memory image of extent header of data section $D_1$, with extent size of $u$. The N-hybrid reuses only the partial extents whose remaining number of free blocks is more than half of total blocks in an extent, to prevent widespread file allocation across partial extents.

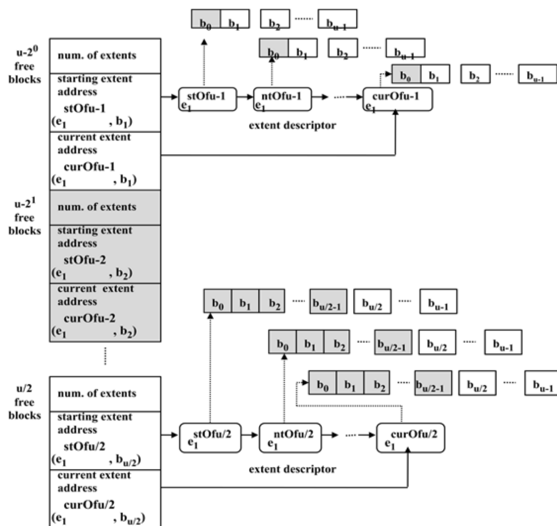


Fig. 4 Extent header of data section $D_1$ where the extent size is $u$

The extent header is organized with $log(u/2)+1$ number of header entries such that each entry points to the linked list of partial extents with $u\text{-}2^i$ number of free blocks, where $0 \leq i \leq log(u/2)$. Each header entry consists of three attributes. The number of partial extents linked at the header position leads and the starting and current extent addresses of the associated linked list follow. In the first header entry, the insertion to the list occurs at the current extent address, $(e_1^{curOfu-1}, b_1)$, and the deletion due to extent reuse occurs at the starting extent address, $(e_1^{stOfu-1}, b_1)$.

*B. Algorithm for SSD Extent Allocation*

When a new file with size of $f$ is created in SSD partition, N-hybrid first chooses a data section $D_i$ in such a way that minimum number of extents are needed to allocate the file. After determining the appropriate data section, N-hybrid checks the extent bitmap of the data section to allocate clean extents.

If the entire blocks of the last extent are not used and the remaining number of free blocks, $w$, is more than or equal to $u/2$, then the extent descriptor of the last extent is inserted to the $i^{th}$ header position such that $u\text{-}2^i \leq w < u\text{-}2^{i\text{-}1}$. If there are not enough clean extents available, then N-hybrid looks for the partial extents by checking the extent header.

With a file size $f$ that needs less than or equal to $u\text{-}1$ number of free blocks, N-hybrid reuses a partial extent whose unused free space is closest to $f$. With a file size larger than $(u\text{-}1)*b$, N-hybrid checks from the first header position until enough number of partial extents are obtained for the file.

The file metadata and data in SSD and HDD partitions are connected by SSD associated attributes in the inode. The *SSD_active* indicates whether the corresponding data are available in SSD partition. This flag is turned off when the cached data are evicted from SSD address space by the extent replacement algorithm.

The *SSD_wr_done* indicates the atomic write between HDD and SSD address spaces. The flag is enabled only when the file is safely stored in both address spaces. The *SSD_section* contains the ID of SSD data section where the associated file is allocated and the *SSD_address* is a sequence of extent addresses consisted of extent number, starting block address and block count.

*C. File I/O Operation*

The selective file mapping enables to assign files to a specific data section where I/O cost related to those files can be optimized, such as assigning large files to a data section with large-size extent. The selective file mapping between directory hierarchy and a specific SSD data section is defined in the configuration file.

When N-hybrid is mounted, the in-core map table is constructed to index the associated data section. This map table contains the map information specified in the configuration file and returns the section descriptor which will be stored in the corresponding directory inode. The section descriptor is inherited to all the descendants of the mapped directory.

The file creation is initiated by allocating the necessary file metadata in HDD address space. The file write operation on both HDD and SSD address spaces is simultaneously performed. As stated earlier, new files are allocated in the appropriate SSD data section, by referring the extent bitmap and header. Simultaneously, allocating files to HDD address space is performed by using ext2/3 allocation modules which are performed based on block group [12].

After the file allocation in SSD address space is completed, a sequence of extent addresses are assigned to file inode, and *SSD_active* and *SSD_wr_done* are enabled to indicate available cached data. The file read operation is started by checking

*SSD_active* to see if the associated file is cached in SSD partition. In case that *SSD_active* and *SSD_wr_done* are both enabled, the cached data stored in *SSD_address* are brought into memory. Otherwise, the file stored in HDD address space is brought into memory and also is updated to SSD address space.

The N-hybrid provides a circular, multilevel LRU queue per data section for SSD extent replacement. In the circular LRU queue, the hotness of a file is determined by file access time, meaning that file inode is linked to the queue, according to the increasing order of access time. The queue pointed to by *current_level* contains hot files and the queue pointed to by *flush_level* contains cold files to be flushed out, in case that there are insufficient free extents available. In the current implementation, each queue links 1K number of accessed inodes.

Once $i^{th}$ queue is full, the recently accessed inode is inserted to the beginning of $((i+1) \bmod n)^{th}$ queue. The re-referenced inodes are moved to the queue pointed to by *current_level*, with their indices being modified to reflect the movement between queues.

When available SSD memory in a data section drops below a threshold, the files starting from the queue pointed to by *flush_level* are flushed out from SSD partition until sufficient free SSD extents are obtained. In such a case, because their copies are available in HDD partition, only thing to be done for SSD flush is to turn off *SSD_active* and *SSD_wr_done* in the corresponding inodes.

## IV. Performance Evaluation

By using Bonnie++ file system benchmark, we compared I/O performance of N-hybrid with that of two file systems: ext2 [12] and xfs [13].

For the evaluation of small files, we changed the chunk size from 1Kbytes to 16Kbytes. Fig. 5 shows I/O performance of sequential write operations. As can be seen in the Fig. 5, in case of 16Kbytes of chunk size, N-hybrid outperforms other file systems, even better than the other file systems installed on the SSD device. This shows that the variable-length of extent size works well on small-sized files.

In both ext2 and xfs, the performance on the SSD device is much better than that on the HDD device. It is noted that, due to the limited storage capacity and high cost, it is not reasonable to store a large amount of data, on top of ext2 or xfs file systems installed on the SSD device.

Fig. 6 shows I/O performance of sequential rewrite operations. In case of sequential rewrite operations, the performance of N-hybrid is almost similar to xfs, but is much higher than ext2. This is because, in case of rewrite operations, the file metadata operations which need to access the HDD partition of N-hybrid does not occur frequently. In other words, most accesses in N-hybrid take place in the SSD partition.

Fig. 7 shows I/O performance of sequential read operations with the impact of memory cache. In this evaluation, the performance of N-hybrid is much similar to those of two other file systems installed on both devices. Based on the evaluation, we can conclude that reading N-hybrid file metadata from the HDD partition does not deteriorate the entire I/O performance.

Fig. 8 shows the performance results for sequential file creations. As can be seen in Fig. 8, the performance of xfs on the SSD partition is much higher than the other file systems, even higher than N-hybrid. We believe that the extent-based file allocation using B++ tree is very effective on the SSD device. Even though N-hybrid supports variable-length extent allocation, it should perform the file metadata and data operations on the HDD partition. This might lower the file allocation performance of N-hybrid. However, storing large-scale files on the SSD device is very costly. Therefore, even though the performance of xfs file allocation on the SSD device is excellent, it is not reasonable to store such a large-scale file data, on top of xfs installed on the SSD device.

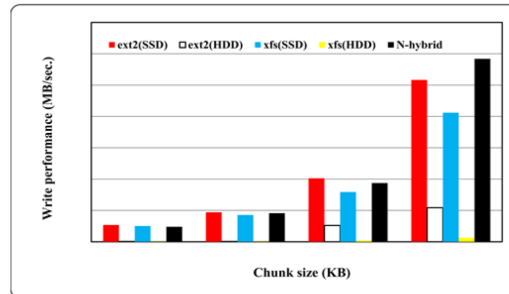


Fig. 5 Sequential write performance

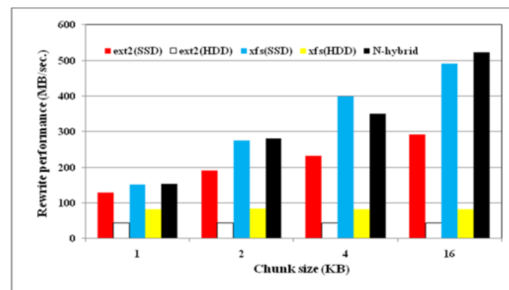


Fig. 6 Sequential rewrite performance

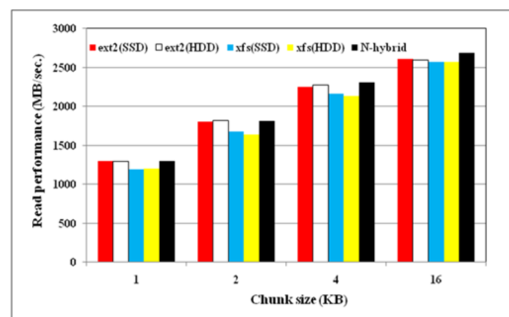


Fig. 7 Sequential read performance

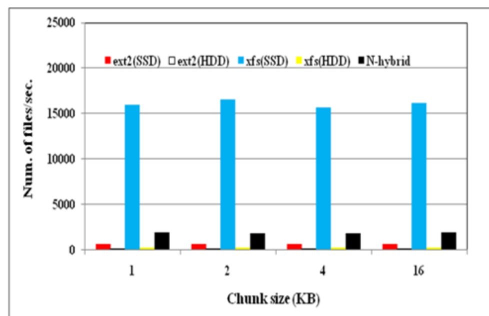


Fig. 8 Sequential create performance

## V. Conclusion

Even though many researchers recognize SSD strengths, such as high I/O performance and reliability, SSD usage in real products has been extremely limited to small-size memory devices. 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 storage capacity. The integration of both devices was performed by providing a flexible internal structure which enables to utilize excellent sequential performance of existing file systems.

The performance evaluation shows that achieving high I/O performance by combining the potential advantages of both SSD and HDD devices is possible. The strength of N-hybrid is most noticeable when its write performance is compared to the corresponding performance of both ext2 and xfs on HDD device. Such a performance speedup is achieved by integrating SSD write-through cache. The write experiment using Bonnie++ benchmark indicates that the mechanical moving overhead of HDD more affects write performance than the semiconductor overhead of SSD. On the contrary, the read experiment demonstrates that, in addition to the use of memory cache, sequential data coalescing makes it possible for HDD devices to achieve comparable read performance to SSD devices. As a future work, we plan to evaluate N-hybrid with real applications to verify its performance effectiveness.

## Acknowledgment

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

## References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse and R. Panigrahy, “Design Tradeoffs for SSD Performance,” *In Proceedings of USENIX Annual Technical Conference*, 2008, pp.57-90.

[2] A. Rajimwale, V. Prabhakaran and J.D. Davis, “Block Management in Solid-State Devices,” *2009 USENIX Annual Technical Conference*, 2009.

[3] C. Lee, S. H. Baek, K. H. Park, “A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices,” *IEEE Transactions on Computers*, vol. 57, July 2008.

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

[5] J.-W. Hsieh, L.-P. Chang and T.-W. Kuo, “Efficient Identification of Hot Data for Flash-Memory Storage Systems,” *ACM Transactions on Storage*, vol. 2, 2006.

[6] A. Olson and D. J. Langlois, “Solid State Drives – Data Repliability and Lifetime,” *White Paper. Imation Corporation*, 2008.

[7] C. Park, W. Cheon, Y. Lee, M-S. J, W. Cho and H. Yoon, “A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications,” *18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP’07)*, 2007.

[8] J. Kim, J-M. Kim, S-H. Noh, S-L. M and Y. Cho, “A Space-Efficient Flash Translation Layer for Compact Flash Systems,” *IEEE Transactions on Consumer Electronics*, vol. 48, May 2002.

[9] Intel Corporation, “Understanding the flash translation layer (FTL) specification,” *Technical Report*, Dec. 1998.

[10] M. Saxena and M. Swift, “FlashVM: Virtual Memory Management on Flash,” *In Proceedings of USENIX Annual Technical Conference*, Boston, MA, 2010.

[11] E. Gal and S. Toledo, “Algorithms and data structures for flash memories,” *ACM Computing Surveys (CSUR),* vol. 37, June 2005.

[12] 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.

[13] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Teck, “Scalability in the XFS File System,” *In Proceedings of the USENUX 1996 Technical Conference*, San Diego, USA, 1996.