RAFS: A RAID-Aware File System to Reduce the Parity Update Overhead for SSD RAID

Chenlei Tang[†], Jiguang Wan[†]*, Yifeng Zhu[‡], Zhiyuan Liu[†], Peng Xu[†], Fei Wu[†], Changsheng Xie[†]
[†]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
[‡]Tianjin Chengjian University, University of Maine
{chenleitang, jgwan, zhiyuanliu, xp04193, wufei, cs_xie}@hust.edu.cn, Yifeng.Zhu@maine.edu

***Abstract*—In a parity-based SSD RAID, small write requests not only accelerate the wear-out of SSDs due to extra writes for updating parities but also deteriorate performance due to associated expensive garbage collection. To mitigate the problem of small writes, a buffer is often added at the RAID controller to absorb overwrites and writes performed to the same stripe. However, this approach achieves only suboptimal efficiency because file layout information is invisible at the block level.**

This paper proposes RAFS, a RAID-aware file system, which utilizes a RAID-friendly data layout to improve the reliability and performance of SSD-based RAID 5. By leveraging delayed allocation of modern file systems, RAFS employs a stripe-aware buffer policy to coalesce writes to the same file. To reduce parity updates, RAFS compacts buffered updates and flushes back in stripe units to mitigate the parity update overhead. RAFS adopts a stripe-granularity allocation scheme to align writes to stripe boundaries. Experimental results show that RAFS can improve throughput by up to 90%, compared to Ext4.

I. Introduction

Solid-state drives (SSDs) are widely used in modern computer systems as primary data storage. In comparison with hard disks, SSDs suffer from a limited amount of write endurance. A flash cell can only sustain a limited number of write operations known as program/erase (P/E) cycles. To improve SSD capacity and drop the cost per bit, manufacturers continuously shrink flash memory cells into smaller geometries or enforce each cell to represent more bits. Unfortunately, both scaling technologies have side effects on the endurance [3], [6]. As the amount of charge stored in each cell decreases in the newer generations of SSDs [6], the probability of error bits increases significantly, leading to diminishing reliability. Although an SSD controller uses sophisticated error-correcting codes (ECCs) internally to correct bit errors, tolerance to whole-device failures is also of great importance. This is because blocks within an SSD tend to wear out at the same rate due to internal wear leveling techniques [4]. Therefore, device-level redundancy is important for an SSD array.

SSD RAIDs are used for applications that demand for high performance and high reliability. By leveraging parity or redundancy, data in a bad block or on a faulty device can be recovered. For every data write, the corresponding parity should also be updated, which aggravates the write endurance issue. For example, a small write to RAID-5 incurs four I/Os, including two reads and two writes. For SSDs, these extra writes also increase the garbage collection overhead. To address these problems, a buffer is often used at the RAID controller to absorb writes performed to the same stripe and overwrites [5], [8], [9], [11]. However, the RAID controller has no knowledge of high-level data layouts. For instance, it does not know whether data blocks belong to the same file. Therefore, when allocating space for new data, it treats all allocation requests the same, without taking the implication of parity updates into consideration. Furthermore, to bridge the processor-storage performance gap, the buffer cache is used in both the file system and the RAID controller. The effect of reducing parity updates is not commensurate to their aggregate buffer cache size. The reason is the buffer in all modern file systems does not explicitly handle the write endurance issue, even though it has great opportunities because dirty data may remain in the file system buffer as long as a few seconds [2]. Although there are some file systems [10], [14] for flash, they are not designed for SSD RAID, which don't take parity updates into consideration.

In this paper, we propose RAFS, a RAID-aware file system, which takes advantage of the RAID architecture and the file layout information and leverages the delayed allocation feature of modern file systems to efficiently coalesce writes that go to the same stripe. The goal is to reduce parity updates and mitigate the write endurance problem and the garbage collection overhead for SSD RAIDs. Our idea is motivated by the following observations. First, storage allocation in the logic block address space is initialized and performed by file systems. Therefore, file systems can best observe high-level I/O behaviors and thus have the best opportunity to align writes to stripe boundaries via smart space allocations. Second, write requests exhibit strong spatial and temporal locality at the file level. As shown later in this paper, within a short time interval of 100 *ms*, we observe that up to 40% of writes go to the same file.

Specifically, We make the following contributions in this paper.

- We incorporate RAID organization into file system design and propose stripe-friendly file data layouts, which organize file data into whole stripe space to reduce the chance data being physically scattered into different stripes as file systems age.
- We design a new stripe-aware buffer management policy, which judiciously leverages the delayed allocation

* Corresponding author: jgwan@hust.edu.cn

978-3-9819263-2-3/DATE19/©2019 EDAA

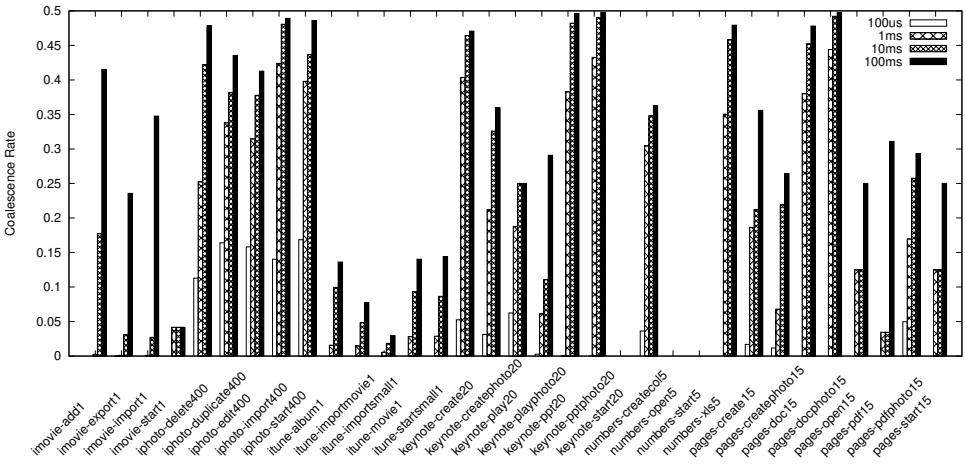


Fig. 1. Coalescence rate of iBench

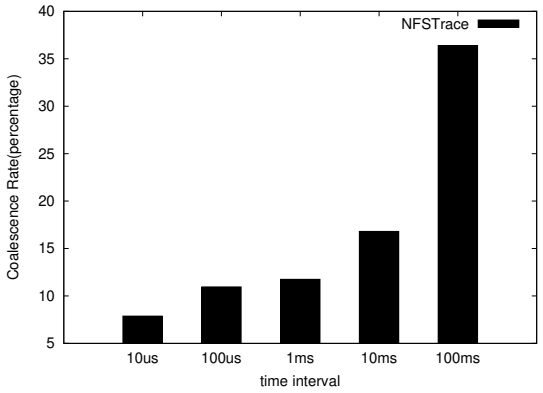


Fig. 2. Coalescence rate of NFS trace

function in modern file systems to delay and coalesce small writes, resulting in less parity updates. Small write requests are compacted into stripe units and later are flushed to SSDs together to eliminate small writes.

- We propose a stripe-granularity allocation scheme to allocate in stripe units to align writes to stripe boundaries.
- We implement our design based on Ext4 and evaluate it under various representative workloads. The results show that RAFS outperforms modern file systems significantly.

II. Background and Motivation

A. Flash-based SSD and RAID

SSD RAIDs are widely deployed as the back-end storage for enterprise servers. In industry, parity-based RAIDs (such as RAID-5) are the most popular due to their superior tradeoff between capacity and added data availability. They benefit from combining multiple drives into a single one to provide high performance as well as redundancy. By leveraging parity, data can be recovered from device failures. However, parity is a double-edged sword. It induces extra writes to update the parity when data is written. This issue is called *write amplification*. What's worse, frequent parity updates can wear flash chips out more rapidly. Hence, parity updates in RAID not only influence the performance, but also diminish the lifetime of SSDs.

B. Data Layout and Write Amplification

Most file systems are not aware of the internal structure of RAIDs. They simply treat the underline RAID as a block device and allocate space in block-granularity. As a result, they do not mitigate write amplification. For an in-place update file system, it updates a file in a read-modify-write way and data must be written to the original location in the file. It is not friendly to SSD RAIDs due to parity updates. For a copy-on-write file system, it writes updates to a new copy. It is easy to achieve atomic operations and consistency, but high write amplification is its drawback. For a log-structured file system, it writes data in a sequential order to fully exploit the bandwidth of the storage device, but garbage collection often causes significant performance overhead.

Both file data and file system metadata update operations cause write amplification in RAID due to parity updates. However, these updates are derived from different access patterns and have different characteristics. Generally, metadata updates are smaller than file data updates. And metadata is updated more frequently to maintain consistency after a crash. Write amplification caused by file data updates could be mitigated by caching in two scenarios: small file updates and significant file updates. For small files, updates are typically small and easy to induce large write amplification. Using a cache to coalesce is an alternative approach to reduce write amplification. For large files, data within a file is often accessed together due to spatial locality. Coalescing updates to a large file can reduce write amplification in RAID.

C. Delayed Allocation

In conventional file systems, such as Ext3, space is allocated for a file whenever the system call *write()* is called. When a file is deleted, file systems immediately reclaim its space. As

files are created, moved, appended to, truncated, and deleted, data of a file tends to be scattered on storage devices, causing data fragmentation.

Delayed allocation is a mechanism used in most modern file systems, such as Ext4, XFS and BtrFS, to reduce fragmentation. In the delayed allocation, data blocks are not allocated until the buffer cache is flushed. It allows file systems to buffer updates in memory and allocates continuous space that fits the file size, leading to less fragmentation. Delayed allocation coalesces small writes. If small writes go to the same stripe, extra parity updates can be eliminated.

What is the impact of delayed allocation on write requests in real workloads? We measure the coalescence rate of I/O traces (including iBench [7], NFS Trace [1]) in different amounts of delayed time, as shown in Figure 1 and 2. The *coalescence rate* is defined as the percentage of writes that go to the same stripe during a given time interval(no bar means 0). The iBench consists of many productivity and multimedia application traces. The NFS trace was collected from large NFS servers. The NFS trace is too large (at a magnitude of TB) and we randomly select a sub-trace (about 16 GB) to analyze.

Before allocation takes place, writes going to the same stripe are *coalescent*. Although the exact delay time of allocation is determined dynamically by the buffer, we can use a simple strategy to evaluate the effect of delayed allocation by fixing the delay to different values. In iBench, more than 30% traces can achieve a coalescence rate over 40% when the delay time is set to 100 *ms*. In the NFS workload, the coalescence rate can be larger than 35%. In a typical file system, dirty data can remain in the buffer for a few seconds before it is flushed (e.g., 5 seconds for Ext4) [2]. This simple study shows that it is promising to exploit the delayed allocation mechanism of modern file systems to mitigate the write endurance issue for an SSD RAID.

III. RAFS Design

RAFS aims to reduce parity updates by leveraging delayed allocation in modern file systems. To achieve this goal, RAFS deploys four techniques:

- **RAID-friendly layout** to organize file data into whole stripes and reduce the chance of scattering file data into different stripes.
- **Stripe-aware buffer** to delay and coalesce writes to the same file, resulting in less parity updates.
- **Stripe-aware flush** to write data back to RAID SSDs in stripe units to reduce the overhead of parity updates.
- **Stripe-granularity allocation** to allocate in stripe units to align writes to stripe boundaries.

A. Overview of RAFS

Figure 3 shows the file system data layout of RAFS. In RAFS, the logical address space in the file system is divided into multiple stripe groups. A stripe group is comprised of a set of consecutive stripes. A stripe is the basic unit of allocation, which contains multiple pages. In this paper, we use the concept stripe to indicate the stripe in RAFS instead of the stripe in RAID without specific description (difference shown in Figure 3).

RAFS divides stripe groups into two types: small stripe groups for small files and large stripe groups for large files. We classify the file type by comparing with the size of a stripe. In the small stripe group, it adopts an out-of-place-update strategy to improve the performance of small files access. In the large stripe group, it takes an in-place-update manner to ensure sequentiality in large files at file system level. However, garbage collection will degrade the performance significantly once free space is nearly exhausted in small stripe groups [14]. Thus RAFS maintains bitmaps to explicitly indicate the validity of data pages. In consequence, RAFS can directly allocate space in small stripe groups by checking bitmaps without garbage collection at file system level.

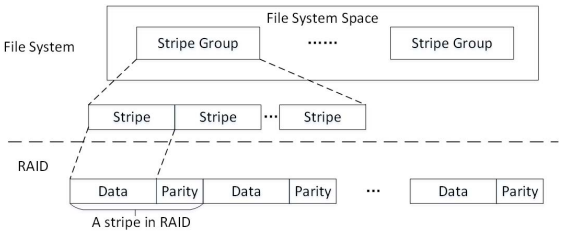


Fig. 3. RAFS data layout

B. Stripe-Aware Buffer Policy

To mitigate the write amplification induced by parity updates, RAFS buffers updates by leveraging delayed allocation so that data in the same file can be aggregated together. We design a stripe-aware buffer policy inspired by the *buddy algorithm* in Ext4.

The *buddy algorithm* is illustrated in Figure 4. It splits all page frames into 11 linked-lists. Each entry in the n^{th} linked-list points to 2^n consecutive page frames. It allocates space in a unit of 2^n page frames (e.g., 1, 2, 4, 8, ..., 1024 page frames). Entries in different granularities of 2^n will be written to the same block group to keep sequentiality if there is enough consecutive space. However, it will waste space if the request size is smaller than the largest granularity (i.e., 4MB). For instance, if the request size is 40 pages, the buddy algorithm will allocate 2^6 pages and 24 free pages are wasted. Buddy algorithm can coalesce data to save space in buddy blocks. However, the buddy blocks are defined as two blocks with the same size and consecutive addresses. Only adjacent blocks can be regarded as buddy blocks. Hence, it will waste space when the number of requested page is small and not in a unit of 2^n.

To overcome the drawback of *buddy algorithm*, we propose a stripe-aware buffer policy. In this policy, we maintain multiple Buffer Linked-lists (*BL*s). Each *BL* contains entries in the granularity of 2^n pages (e.g., *BL 0* stores entries consisted of 2^0 pages). A request can be split into several sub-requests. For example, when applications issue a write request of 47 pages, the request can be split into multiple basic units of

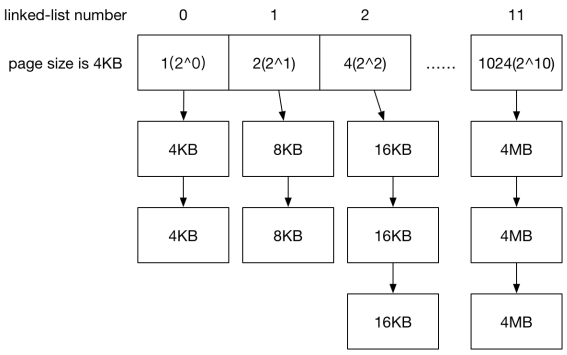


Fig. 4. Buddy algorithm

2^n(e.g., 47 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0). Each sub-request will be stored in its corresponding *BL*. When RAFS begins to flush data, it will flush in stripe units if there is enough consecutive space. Otherwise, it will select entries from all *BL*s to organize them together according to available space in RAID. Thus it is efficient to assemble small files together. RAFS splits a request into multiple sub-requests for better allocation, but makes data scattered in DRAM. Hence, to improve read performance in DRAM, RAFS uses a B+ tree to index buffered data for each file. The B+ tree has been widely used in databases and file systems due to its efficiency of indexing sparse data. The root pointer of the B+ tree is stored in the VFS inode of the file. The key of the leaf node is the logical offset within the file. The value of the leaf node stores the BL number and the offset within this BL.

C. Stripe-Aware Flush Policy

To ensure data persistence, we use a background thread to flush buffered data back into RAID. There are several scenarios in which the buffer must be flushed. The following gives a few examples.

1) There is no enough free space in buffer. When the total free space is lower than a given threshold T_f(e.g., 20%), the thread begins to flush data.
2) Dirty data has not been flushed to SSDs for longer than a time threshold T_t(e.g., 5s). This is to meet the consistency requirements.
3) Applications force to flush dirty pages. For instance, the *fsync* command is called to enforce persistence.

When flushing back to RAID, RAFS needs to determine which stripe to write. If the selected stripe is a free stripe, RAFS simply compacts buffered data in stripe units and writes back. Otherwise, RAFS needs to find the longest consecutive valid pages in dirty stripes, then writes them back. It may take extra overhead. But we construct *AL*s(details in Section III-D). It is efficient to find the best stripe to write.

D. Stripe-Granularity Allocation

Conventional block-based file systems treat storage devices as block devices and allocate space in block units. However, the basic unit of space allocation and deallocation is a RAID stripe in SSD RAIDs. SSD RAIDs will modify data and parity when updating a stripe, which introduces high write amplification, especially for small writes. In consequence, to make the best use of the bandwidth of SSD RAIDs, we propose a stripe-granularity allocation scheme at the file system level. By leveraging delayed allocation of modern file systems, we can compact updates together to reduce small writes.

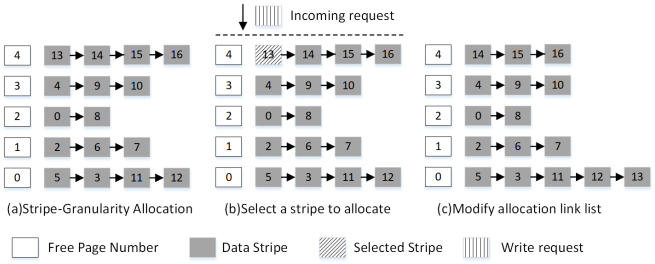


Fig. 5. Stripe-granularity allocation

The stripe-granularity allocation is shown in Figure 5 (a). RAFS maintains multiple Allocation Linked-lists (*AL*s) to record the number of free pages in each stripe. RAFS divides the file system space into stripes and assigns each stripe a unique stripe number. Entries in *AL*s only store the stripe numbers. Thus, *AL*s have a very small memory overhead. Note that stripe is auxiliary for efficient allocation and flush, lookup can still be achieved by mappings stored in inodes. For simplicity, we assume that a stripe consists of 4 pages in this example. Each stripe contains 0-4 free pages. There are two different cases to determine which stripe to be allocated for the current request.

1) For large stripe groups, RAFS simply returns the original stripe number to conduct an in-place update.
2) For small stripe groups, if there is enough free space, RAFS will allocate stripes in the *AL* with 4 free pages to get the best performance. If all free stripes are exhausted, RAFS scans all *AL*s and allocates the stripe with most free pages for this request.

In Figure 5 (b), we assume it is the second case and there are enough free stripes for allocation. Thus, RAFS checks the *AL* with 4 free pages and selects stripe 13 in the *AL* to this request. After determining which stripe to allocate, RAFS needs to update *AL*s to guarantee consistency. Because stripe 13 is allocated, RAFS marks it as a used stripe and moves this entry to the *AL* with 0 free page, as shown in Figure 5 (c). After that, a stripe allocation is completed. If power outage or system failure occurs, RAFS simply scans the bitmap to recover *AL*s.

E. Crash Recovery

It is significant to survive crashes or power failures. RAFS reuses the Linux *jdb2* (journaing block device 2) module to guarantee system consistency. There are three different modes, including write back, ordered data and journal data. Operations can be recorded in journal area for recovery. Once system

crashes or power failure occurs, RAFS recovers committed records and discards uncommitted records by reading the journals. In addition, by scanning the bitmap of each stripe group, RAFS can recover *ALs* to a consistent state. Furthermore, data can be recovered by parities in RAID.

IV. Evaluation

In this section, we compare RAFS with BtrFS [13], F2FS [10] and Ext4 [12]. BtrFS is a copy-on-write file system. F2FS is a log-structured file system optimized for SSD. Ext4 is an in-place-update file system, which is widely used in mobile and enterprise applications. To measure the performance accurately, we disable the journal mechanism in Ext4 and RAFS.

We highlight the comparison in the following three aspects: performance, stripe write traffic (including parity updates) reduced, and scalability with threads.

A. Experimental Setup

We use 5 Intel 535 series 120GB SATA SSDs to construct a software RAID by using *mdadm* tools in Linux. All experiments are conducted on an X86-64 server with 2 Intel Xeon E5606 CPUs and 12GB memory. We compile the 4.1.4 kernel to run all experiments on Ubuntu 16.04. RAFS is implemented based on Ext4 and it is compiled as a module.

To compare RAFS with the other three file systems, we build a tool called *stripe counter* to collect the statistics of stripe writes (including the total number of stripe writes, the total number of full stripe writes, etc.). *Stripe counter* is implemented in the device driver. We modify the RAID module to intercept stripe writes and check whether it is a full stripe write or not for each write request.

Workloads We use 5 workloads for evaluation. Most of them (i.e., Fileserver, Videoserver, Webserver, Webproxy) are generated by Filebench [15] except Postmark [16], which is developed by NetApp.

B. Overall Performance

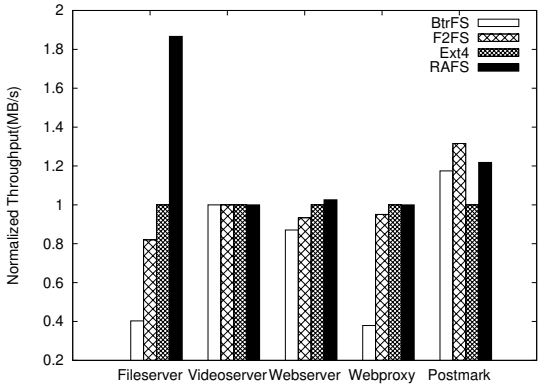


Fig. 6. Overall performance

Figure 6 shows their throughput normalized to Ext4. RAFS outperforms the other three file systems in most cases.

Especially, RAFS can improve the performance up to 86% under the Fileserver workload. This workload generates many random file writes. Data belongs to the same file is buffered respectively in RAFS and then flushed back together to RAID in stripe units. Hence, the throughput of RAFS can be improved significantly. Although Ext4 has delayed allocation, scattered in-place updates result in large write amplification. F2FS uses a log-structured way to append data, but it flushes back when data is not in the same file, which cannot fully utilize cache. Random writes can bring extra write amplification in BtrFS due to copy-on-write.

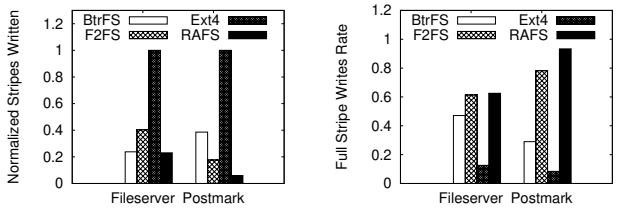
For the Webserver workload, RAFS also outperforms the other file systems. It benefits from delayed allocation and stripe-aware flush. Ext4 outperforms F2FS. That is because Webserver is a read-intensive workload, and files can be cached for further reads.

Under the Videoserver workload, all four file systems achieve the same performance. That is because this workload contains multiple sequential operations for large files. It is difficult to buffer all data in the limited cache. Thus, the performance is determined by the bandwidth of storage devices.

In the Webproxy workload, RAFS and Ext4 get the same performance. Because this workload exhibits strong access locality, the write strategies adopted by RAFS get little impact.

In Postmark, it emulates the workload of mail servers. Append operations comprise a large proportion. F2FS writes in a log-structured manner, thus it can achieve the highest throughput. There are little duplicated writes, RAFS can seldom coalesce to exploit cache.

C. Endurance Evaluation

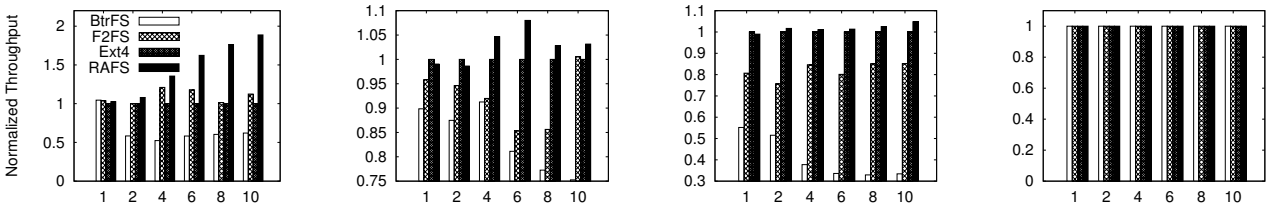


(a) Write traffic (b) Full stripe write rate

Fig. 7. Endurance evaluation

To evaluate the endurance improvement of RAFS, we select two write-intensive workloads. Figure 7(a) presents the normalized write traffic(the number of total stripes written) in these two workloads. RAFS outperforms the other three under both workloads. By leveraging delayed allocation, RAFS can buffer random updates in memory. For small files, RAFS writes updates in a sequential order to avoid write amplification. Further, when the buffer is flushed, RAFS adopts a stripe-aware flush policy to mitigate the write amplification. RAFS can decrease the write traffic up to 94%, 17%, and 32%, when compared with Ext4, F2FS, and BtrFS, respectively.

Figure 7(b) compares the full stripe write rates (the ratio of full stripes written to total stripes written) under Fileserver



(a) Fileserver (b) Webserver (c) Webproxy (d) Videoserver

Fig. 8. Throughput comparison of four workloads under different numbers of threads

and Postmark workload. By comparing full stripe write rates, we can figure out which file system can induce the least write amplification under the same workload. RAFS achieves the highest full stripe write rate under both workloads. Surprisingly, although Ext4 outperforms BtrFS and F2FS in Fileserver, it has the lowest full stripe write rate. The reason lies in the write strategy in Ext4. It writes data to original location in the file, resulting in large write amplification.

Furthermore, the write amplification can be enlarged by parity updates. F2FS gets a high full stripe write rate because it appends data in a sequential way. RAFS buffers writes in memory, then organizes small writes in stripe units to flush back. Thus, it can get a comparable full stripe write rate with F2FS. The results of Postmark are similar to Fileserver except that RAFS has a 10+% higher full stripe write rate than F2FS. But RAFS achieves slightly lower throughput than F2FS in Postmark. One reason for this degradation is that it takes a bit longer time to seek stripes during allocation.

D. Scalability Evaluation

We demonstrate the scalability of RAFS. Figure 8 shows the normalized throughput comparison under different threads. We vary the number of threads from 1 to 10. There are only four workloads beacause Postmark is a single-thread benchmark. RAFS achieves the highest throughput in all workloads. As the number of threads increases, the throughput of four file systems shows a rising trend in most cases except Videoserver. In Videoserver, it shows the same throughput regardless of the number of threads. Because most file operations are sequential. More threads cannot further improve the performance due to full utilization of all available I/O bandwidth. However, in the other three workloads, more threads can effectively improve performance. Especially in Fileserver, RAFS can achieve nearly 1.9x higher throughput than Ext4 when the number of threads is 10. In Webserver, the throughput of four file systems increases when the number of threads goes from 1 to 6. However, it begins to drop slightly when the number of threads increases from 6 to 10. It may be determined by the limited capacity of the cache. Data will be frequently swapped in and out and file systems cannot make full use of the cache.

V. Conclusion

In this paper, we have designed and implemented a RAID-aware file system (RAFS) to reduce the parity update overhead for SSD-based RAID 5. RAFS divides file system space into stripe units for efficient allocation. By leveraging delayed allocation, RAFS buffers updates at the file system level and coalesces writes that go to the same stripe. To further reduce parity updates, RAFS compacts buffered updates and flushes back in stripe units. RAFS employs a stripe-granularity allocation scheme to align writes to stripe boundaries. Evaluations show that RAFS outperforms other modern file systems significantly.

VI. Aknowledgement

This work was sponsored in part by the National Natural Science Foundation of China under Grant No.61472152, No. 61300047, No. 61572209, and No.61432007; the 111 Project (No. B07038); the Key Laboratory of Data Storage System, Ministry of Education; US NSF Grant 1618536.

References

[1] E. Anderson, “Capture, Conversion, and Analysis of an Intense NFS Workload”, in FAST, 2009.
[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, ”Operating systems: Three easy pieces”. Vol. 151. Arpaci-Dusseau Books Wisconsin, 2014.
[3] S. Boboila and P. Desnoyers, “Write Endurance in Flash Drives: Measurements and Analysis”, In FAST, 2010.
[4] L. Chang, “On efficient wear leveling for large-scale flash-memory storage systems”, ACM Applied computing, 2007.
[5] C. Chung and H. Hsu, “Partial parity cache and data cache management method to improve the performance of an SSD-based RAID”. IEEE TVLSI, 2014.
[6] L. M. Grupp, J. D. Davis, and S. Swanson, “The bleak future of NAND flash memory”, in FAST, 2012.
[7] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications”, in SOSP, 2011.
[8] S. Im and D. Shin, “Delayed partial parity scheme for reliable and high-performance flash memory SSD”, in MSST, 2010.
[9] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh, “Improving SSD reliability with RAID via elastic striping and anywhere parity”, in DSN, 2013.
[10] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, “F2FS: A New File System for Flash Storage”, in FAST, 2015.
[11] Y. Lee, S. Jung, and Y. H. Song, “FRA: a flash-aware redundancy array of flash storage devices”, in CODES+ISSS, 2009.
[12] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new ext4 filesystem: current status and future plans”, in Linux symposium, 2009.
[13] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree filesystem”, ACM TOS, 2013.
[14] J. Zhang, J. Shu, and Y. Lu, “ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices”, in ATC, 2016.
[15] “Filebench benchmark,” http://sourceforge.net/apps/mediawiki/filebench.
[16] J. Katcher, “Postmark: A new file system benchmark”, Tech. Rep., Network Appliance, 1997.