Efficient and Consistent NVMM Cache for SSD-Based File System

Youmin Chen, Youyou Lu, *Member, IEEE*, Pei Chen, and Jiwu Shu, *Fellow, IEEE*

Abstract—Buffer caching is an effective approach to improve the system performance and extend the lifetime of SSDs. However, the frequent synchronization operations in most real-world applications limit such advantages. This paper proposes to adopt emerging non-volatile main memories (NVMMs) to relieve the above problems while achieving both efficient and consistent cache management. To this end, an adaptive fine-grained cache (AFCM) scheme is proposed, which is motivated by our observation that the file data in many synchronized pages is partially updated for a wide range of workloads, implying that fine-grained cache management can save the NVMM cache space wasted by the clean parts. To reduce the cache index overhead introduced by fine-grained cache management, AFCM employs a *Hybrid Cache* based on DRAM and NVMM, with which the normal read and write operations are served without performance penalty. We also propose the Transactional Copy-on-Write mechanism to guarantee the crash consistency of both NVMM cache space and file system image. Our experimental results show that AFCM provides up to 84 percent performance improvement and 63 percent SSD write reduction on average compared to the conventional coarse-grained cache management scheme.

Index Terms—Solid state drive, non-volatile memory, file system, cache management, crash consistency

1 Introduction

Flash-based Solid State Drives (SSDs) are electronic storage devices. They don't contain any mechanical parts, and thus provide great advantages in terms of low access latency, low power consumption, compact size, and shock resistance. However, they face the problems of the relatively poor write performance and the limited lifespan. Previous research [1], [2], [3] has shown that excessive write operations not only degrade the SSD-based system performance significantly, but also drastically reduce the lifespan of the SSD device. To relieve this problem, buffer caching is an effective and commonly-used approach to reducing the write traffic of SSDs [4], [5], [6], [7], [8], [9], [10], [11], thereby improving the system performance and extending the SSD's lifetime.

Conventional file systems typically use DRAM as a cache layer to improve the system performance because of its high endurance and low access latency. Unfortunately, the volatile property of DRAM may lead to data loss when the system crashes. Therefore, applications either adopt journaling-based file systems or issue extra synchronization operations (e.g., `fsync` or `fdatasync`) to make the file data persistent [12], [13]. For example, database systems will synchronize the journal file before they can commit a transaction [14]. While the above approaches improve the system reliability, they significantly degrade the effectiveness of the DRAM cache due to frequent accesses to the storage [15], [16].

With the rapid development of emerging non-volatile memory (NVM) technologies, such as 3D-XPoint, phase change memory (PCM), resistive RAM (ReRAM), and memristor, they are promised to provide DRAM-like performance and better endurance than flash memory. Attaching NVM directly to the main memory bus will produce byte-addressable non-volatile main memory (NVMM). Compared with DRAM, NVMM provides the unique characteristics of non-volatile, high density, and low power consumption [17], [18]. Therefore, using NVMM as a persistent cache to improve the performance and endurance of SSDs is practical and effective [7], [8], [9], [10], [11], [19], [20].

As current NVMM devices are of high price and low capacity, one of the key challenges of designing an NVMM cache for SSDs is how to improve the cache utilization. However, existing NVMM-based caching strategies for SSDs manage the NVMM cache space in flash page granularity [7], flash block granularity [8], [9], [10], or mixed page and block granularity [11]. These algorithms mainly focus on exploiting the spatial locality to improve the sequentiality of write-back operations. However, they suffer from buffer pollution and early eviction issues due to coarse-grained management, leading to low buffer utilization and low hit ratio, thereby degrading the system performance.

In this paper, we observe that a wide range of workloads contain a large number of synchronization operations (see Section 2.2). To eliminate the impacts of these synchronization operations, we propose to introduce NVMM cache to buffer these operations, in order to reduce the write traffic to SSDs. To improve the cache utilization, we further propose an Adaptive Fine-grained Cache Management (AFCM) policy

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: chenym16@mails.tsinghua.edu.cn, {luyouyou, shujw}@tsinghua.edu.cn, chenp732@gmail.com.*

Manuscript received 2 Mar. 2018; revised 15 Aug. 2018; accepted 3 Sept. 2018. Date of publication 12 Sept. 2018; date of current version 17 July 2019. (Corresponding author: Jiwu Shu.)
Recommended for acceptance by Y.-H. Chang, J. Ju, and M. B. Tahoori.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2870137

for the NVMM cache. Instead of buffering the whole synchronized page, AFCM selectively buffers the dirty parts of synchronized pages to save the NVMM space wasted by the clean ones. By fully leveraging NVMM's byte-addressability, AFCM improves both the cache utilization and system performance. As far as we know, this is the first work to manage the file data in NVMM with fine-grained granularity. However, there are three key challenges with fine-grained cache management:

1) The overhead of indexing the cached data will become larger with fine-grained management, which may result in low read and write performance due to the increased latency of cache hit/miss judgments.
2) The fine-grained cache management will introduce extra metadata in NVMM, which enables the system to track the data in the recovery process after the system crash (in Section 3.3). However, the above approach may degrade the cache utilization when the synchronized pages are of a high dirty ratio (e.g., close to 100 percent).
3) Leveraging the byte-addressability of NVMM, the fine-grained cache management makes it more complicated to guarantee the NVMM cache consistency. This is because the file pages are fragmented and spread in NVMM cache. Any partial update of the synchronized pages will cause inconsistency of the file system during the system crash.

Accordingly, we make the following contributions:

- With the analysis of different file system workloads, we find that (1) the write operations which need to be explicitly synchronized to the stable storage account for a large portion of the total writes, and (2) the average percentage of dirty cachelines in most synchronized pages is much less than 100 percent.
- To fully utilize the NVMM cache space, we propose an *Adaptive Fine-grained Cache Management* scheme, which dynamically chooses the fine-grained or coarse-grained cache mode to buffer the synchronized pages depending on their ratio of dirty cachelines.
- To alleviate the index overhead caused by fine-grained management, a novel *Hybrid Cache* is introduced by combining both DRAM and NVMM. The DRAM cache is managed with page granularity to serve the normal read and write operations without performance penalty, while the NVMM cache is only used to cache the synchronized data for consistency guarantee. We further propose the *Transactional Copy-on-Write* (*TCoW*) mechanism. It introduces a circular log to efficiently preserve the consistency of both NVMM cache and file system image.
- Our experimental results show that AFCM outperforms the traditional coarse-grained cache management policy by up to 84 percent. Meanwhile, it also achieves 63 percent SSD write reduction on average compared to conventional approaches, thereby significantly extending the SSD's lifetime.

The remainder of this paper is organized as follows: Section 2 introduces some backgrounds and discusses the motivation. We present the design and implementation of AFCM in Sections 3 and 4, respectively. We then present the evaluation results of AFCM in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Background and Motivation

In this section, we first analyze existing cache policies in file systems. We then collect the percentage of `fsync` bytes of different workloads to reveal the synchronization overhead. We also analyze the distribution of dirty cacheline percentage in synchronized pages to show the potential advantages of AFCM. The overhead of crash consistency in file systems is also illustrated in this section, which motivates us to design TCoW.

2.1 Cache Management in File Systems

Existing computer systems are equipped with different storage devices, such as register, DRAM, flash memory, and hard disk. They show different properties both in access latency and bandwidth. As a result, buffer caching is a widely-used approach to fill the performance gap among different storage layers. Cache efficiency is determined by the *Average Cache Access Time* (denoted as T). It depends on the *Cache Hit Access Time* (denoted as T_{hit}), the *Cache Hit Ratio* (denoted as P_{hit}), and the *Cache Miss Access Time* (denoted as T_{miss}). Thus, the value of T can be represented by the following equation:

$$T = T_{hit} * P_{hit} + T_{miss} * (1 - P_{hit}).$$

To improve the cache efficiency, the system designers can optimize each of the three factors.

Because of the high price and low capacity of current NVMM technology, existing cache optimizations mainly focus on designing NVMM-based caching policies from two aspects: One is to improve the cache hit ratio (P_{hit}). As the value of T_{hit} is much lower than that of T_{miss}, a higher cache hit ratio can result in the lower value of T, indicating a better cache efficiency. Meanwhile, as cache hit ratio increases, the cache write-back traffic will also be reduced, thus leading to the longer lifetime of the SSD device. Existing approaches [21], [22], [23], [24] exploit the temporal locality and access frequency to design efficient cache replacement algorithms, so as to keep the data which are most likely to be accessed again in the cache as long as possible. On the other hand, to reduce the cache miss penalty (i.e., T_{miss}). Some researchers have designed efficient cache write-back algorithms with the consideration of the high sequential write performance of SSD [4], [8], [9]. These algorithms mainly exploit the spatial locality and group resident pages in DRAM/NVMM cache in order to flush dirty cache data to the SSD device as sequential as possible. However, they suffer from buffer pollution and early eviction issues due to coarse-grained management, resulting in low cache utilization and low hit ratio.

Worth noticing, the effectiveness of the above two approaches depends on the workload access characteristics, as both of them exploit the access locality or frequency for their optimization. As a result, the ability of existing cache management schemes is limited, because most I/O intensive workloads are mixed with random and sequential I/O requests. In order to be independent of the workload characteristics, we optimize the cache efficiency based on NVMM from another unique perspective (i.e., cache management

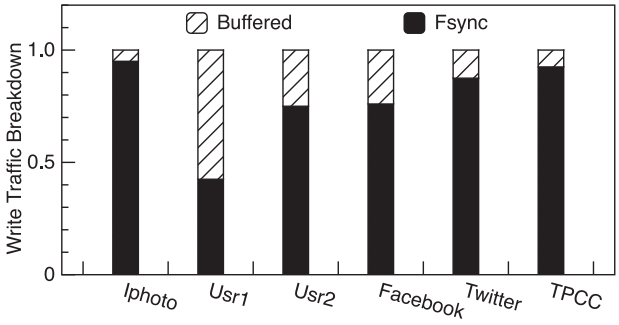


Fig. 1. Percentage of fsync write traffic in different workloads.

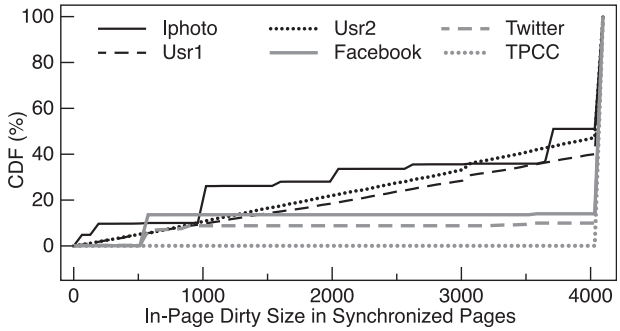


Fig. 2. CDF of in-page dirty size in synchronized pages.

granularity). Since the cache hit ratio is not only influenced by the cache replacement algorithm, but also determined by the total available cache space. Improving the cache utilization leads to higher available cache space, thus providing higher cache hit ratio and system performance.

2.2 Workload Analysis

Previous work [13] has shown that, in most applications, a large proportion of the written data should be explicitly synchronized to the disk in order to ensure their required semantics of data persistency. This will lead to lower cache hit ratio (Section 2.1) even when the cache space is not exhausted. Therefore, the frequency of synchronization operations and the amount of synchronized data can directly impact the system performance and the efficiency of the DRAM cache.

To understand the impact of synchronization operations on the DRAM cache efficiency and the associated system performance, we perform an experiment to track the use of synchronization operations across various workloads. Fig. 1 shows the write traffic breakdown of the *Fsync Write* and *Buffered Write* in these workloads. We can observe that, among the selected six workloads, five workloads have more than 70 percent of the total written data that need to be explicitly synchronized to the storage layer. Moreover, the rest workload (i.e., Usr1) also have nearly 50 percent of the total written data that need to be persisted. To conclude, a large number of applications have a significant portion of the written data with persistence requirements. Thus, adopting NVMM to work as a persistent cache is both necessary and effective.

To learn the potential performance advantages of fine-grained cache management, we further analyze the distribution of in-page dirty ratios in all synchronized pages.[1] The in-page dirty ratio is the fraction of dirty data size of a page in the total page size. The results are shown in Fig. 2. From the figure, we find that the dirty ratio distribution of synchronized pages in different workloads varies dramatically. For example, the TPCC workload has more than 99 percent of total synchronized pages with 100 percent dirty ratio (i.e., all the data in these synchronized pages are dirty). In contrast, the Iphoto, Usr1, and Usr2 workloads have a large proportion of total synchronized pages whose dirty ratio is much less than 100 percent (i.e., only parts of the data in these synchronized pages are dirty). This indicates that conventional cache schemes with coarse-grained management (i.e., in page granularity) tend to make the cache space of NVMM wasted by synchronized pages that are partially updated, thereby degrading the cache space utilization. This observation motivates us to design a novel fine-grained cache management scheme based on NVMM. It achieves this by reducing the wasted cache space caused by partially-update pages, so as to improve the cache utilization and system performance.

2.3 Crash Consistency Overhead in File System

The crash consistency of file systems guarantees the data integrity of upper-layer applications. Existing file systems [25], [26], [27], [28] have developed different techniques to provide crash consistency.

The most popular solution to consistently update the file data is journaling (a.k.a., Write-Ahead-Logging) [29], [30], [31]. Before updating the file data to storage device in place, the corresponding log entry is first made persistent in the log area. However, it introduces double updates to the disk, thus restricting the system performance. Another approach is Copy-on-Write (CoW) [26], [32], which is adopted in many file systems, such as Btrfs and ZFS [33]. Rather than overwriting files in place, it redirects updated data to the new location, and validates it by updating the pointer of file's metadata to the new place. This technique avoids double writes of data, however, any update of file data will propagate to the root inode, leading to the sub-optimal performance.

There are also some NVMM-based cache schemes [19], [20] that are designed with crash consistency for file systems. UBJ [19] adopts NVMM to work as both journaling device and buffer cache. To commit the blocks in NVMM-based buffer cache, they are in-place frozen to act as journal log, and are checkpointed later to the disk to free the NVMM space. However, UBJ needs an extra memory copy when updating the frozen data. It also relies on conventional checkpoint technique to reclaim NVMM space, which is extremely inefficient. Tinca [20] avoids double writes by making the blocks of data in NVMM switch between roles of being committed and checkpointed. However, both of them show limited universality, because of the closely coupled design with commit and checkpoint mechanisms in Ext4. Besides, they suffer wear problems, as all the file I/O will flow through the NVMM cache.

As a whole, existing crash consistency approaches in file system either incur high overhead, or have limited universality. In this paper, we *target at* designing efficient and consistent cache scheme by providing atomic update interface. It makes the underneath file systems naturally inherit the high consistency property, regardless of their implementation.

1. The default page size is 4 KB.

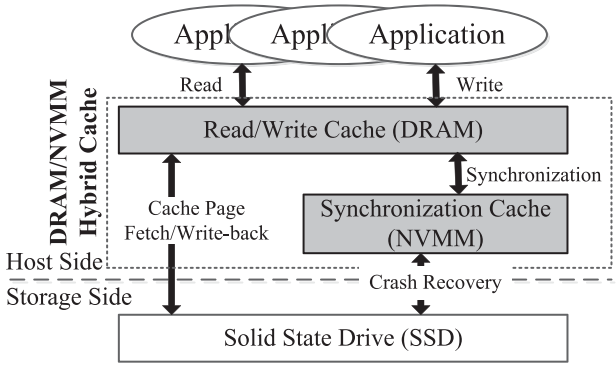


Fig. 3. The hybrid cache architecture of AFCM.

3 AFCM Design

In this section, we will first discuss how to design an efficient fine-grained cache management scheme based on NVMM, so as to improve the system performance and reduce the write-back traffic to SSDs. We then present the design of transactional copy-on-write mechanism which guarantees the system consistency by providing atomic synchronization operations.

3.1 Hybrid Cache Architecture

While fine-grained cache management policy can improve the cache utilization (mentioned in Section 2.2), it also increases the overhead of the cache index, leading to lower read and write performance. In other words, the finer the cache granularity, the larger overhead of indexing the cached data. To overcome this problem, we propose a *Hybrid Cache* based on DRAM and NVMM. The design of this architecture is shown in Fig. 3.

The SSD's cache layer consists of a volatile cache (i.e., DRAM) and a non-volatile cache (i.e., NVMM), both of them are on the host side. The NVMM is attached directly to the memory bus alongside DRAM and accessed via memory interfaces (i.e., Load/Store instructions) so as to support byte-addressability. In addition, the NVMM cache space is managed via a fine-grained way that buffers the synchronization operations by exploiting its non-volatility. To avoid the impact of fine-grained cache management on the performance of normal read and write operations, the extra DRAM cache is managed with page granularity to buffer the read and write operations so that the index overhead is small.

When serving the normal read and write operations, the page fetch and write-back operations are directly issued between the DRAM cache and back-end SSD device. Differently, the NVMM cache is mainly used for ensuring the data persistence required by applications. When the file system receives a synchronization operation, AFCM will flush the related DRAM pages which need to be synchronized to the NVMM cache in a fine-grained manner (see Section 3.2 for more details). This design eliminates the write traffic to SSD caused by synchronization operations in traditional DRAM-only cache environment, thereby improving the system performance and extending the SSD's lifetime. Besides, AFCM only allows the synchronized data flow to the NVMM device, thus reducing the wear on NVMM when compared with existing cache schemes [19], [20] that make all file data flow through the NVMM.

The cache eviction of both NVMM and DRAM cache only occurs when there are no free blocks. To replace a dirty page in the DRAM cache to the SSD device, the associated data in the NVMM cache will be discarded as these data are already persisted to the SSD device. When a replacement operation occurs in the NVMM cache, the replaced cache data should also be written back to SSD device before they are evicted from the NVMM cache in order to guarantee the data persistence. For simplicity, the replacement for it is page-oriented just like DRAM cache (in Section 3.2.1). To retain the benefit of existing cache replacement algorithms for normal read and write operations, the replacement in the NVMM cache will not alter the priority of the corresponding page in the DRAM cache, but change its state from *dirty* to *clean*. The reason is that this DRAM page may still have a high caching priority even though the corresponding data are evicted from the NVMM cache. As a result, the newly added NVMM cache layer of our design will not change the existing DRAM cache policy, but is mainly used to reduce the write traffic to SSD caused by synchronization operations as much as possible.

Discussions. Previous works pointed out that NVMM also has endurance problem. However, this is not to be an issue in AFCM because 1) AFCM separates the I/O path of asynchronous/synchronized data, so the write traffic between the NVMM and SSD devices is rebalanced; 2) Multiple dirty data blocks in the same file will be merged and compacted in NVMM, so the write traffic to the SSD devices will be extremely reduced; 3) We only need to modify the dirty parts of the data blocks in NVMM, which further limits the write amplification to the NVMM. In summary, our proposed hybrid architecture is capable of reducing the write traffic to NVMM and SSD compared with the "single-level architecture" with only SSD device equipped.

3.2 Adaptive Fine-Grained Cache Management

We will first introduce the simple fine-grained cache scheme (SFCM, Section 3.2.1) that is used to improve the cache utilization. Based on that, an adaptive cache management (AFCM, Section 3.2.2) is proposed to dynamically choose the cache mode (cacheline or page granularity) according to the dirty ratio of each synchronized page.

3.2.1 Design of SFCM

The design of the SFCM scheme is shown in Fig. 4. As the minimum interaction granularity between CPU and memory is the CPU cacheline size (i.e., 64 B), SFCM would not buffer the whole synchronized pages in NVMM, but only writes the dirty cachelines to the NVMM cache in order to improve the cache utilization of the NVMM. Besides, the corresponding DRAM pages will not be discarded or replaced after the completion of the synchronization operation. Thus, all the latest data still remains in the DRAM cache to service the normal read and write operations.

To facilitate the system to judge whether a dirty cacheline is already located in the NVMM cache or not during a synchronization operation, SFCM maintains a linked-list based index for each logical file page, the head of which is indexed by a global in-memory hash table (see Fig. 4). To save the storage overhead of the index and improve the indexing

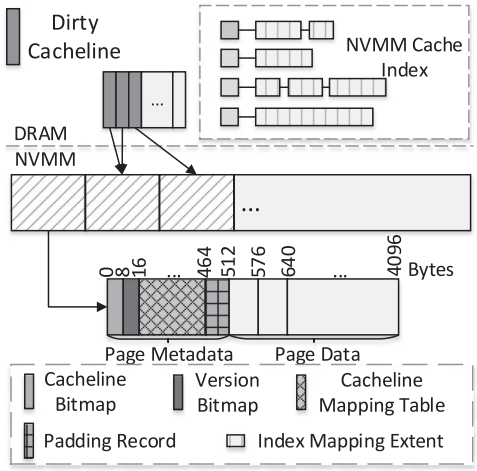


Fig. 4. The design for the SFCM scheme.

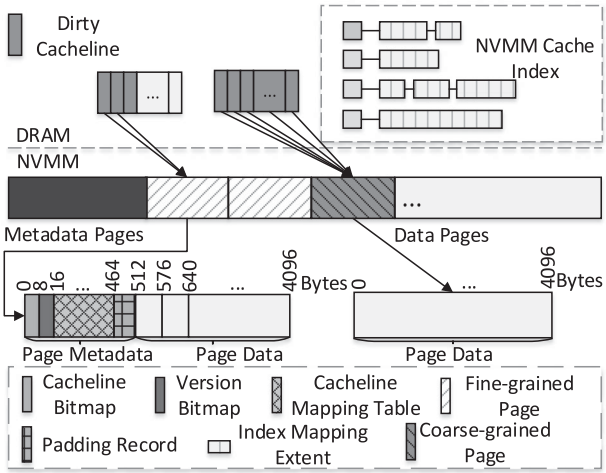


Fig. 5. The design for the AFCM scheme.

performance, each entry in the linked list represents a continuous mapping extent. Each mapping extent is a triple tuple, denoted as $\langle \text{Logic\_File\_Addr}, \text{Physical\_NVMM\_Addr}, \text{Mapping\_Len} \rangle$, indicating that the continuous logical file interval from Logic_File_Addr to $(\text{Logic\_File\_Addr} + \text{Mapping\_Len})$ is mapped to a continuous physical interval from $\text{Physical\_NVMM\_Addr}$ to $(\text{Physical\_NVMM\_Addr} + \text{Mapping\_Len})$ in the NVMM cache. Since SFCM writes the synchronized pages to the NVMM cache based on the cacheline granularity, the value of each item in the triple tuple is aligned to the cacheline size. To facilitate the search and merge operations of the mapping extents, all the mapping extents belonging to the same logical page are sorted by Logic_File_Addr. For each dirty cacheline that needs to be synchronized to the NVMM cache, SFCM first refers to the *NVMM Cache Index*. If it's a hit, SFCM writes this cacheline to the corresponding NVMM location directly. Otherwise, SFCM should allocate new cache space from NVMM with cacheline granularity and update the corresponding cache index.

To identify the corresponding SSD location of each buffered cacheline in the NVMM during the system's crash recovery process, SFCM needs to maintain an extra mapping table for each buffered cacheline in the NVMM. As shown in Fig. 4, SFCM partitions the NVMM cache space to continuous pages, and the space within each page is allocated based on cacheline granularity. The page metadata is located at the head of each NVMM page, which consists of four parts: The first 8-bytes *Cacheline Bitmap* is used to identify that whether a cacheline in the page data area is in *valid* or *free* state. The subsequent *Version Bitmap* is used for crash consistency, which will be introduced in Section 3.3. *Cacheline Mapping Table* is used to record the corresponding SSD address of each cacheline in the page data area. The *Padding Record* is used to make the page metadata area align to the cacheline size.

Though the space allocation of the NVMM cache is based on the cacheline granularity, the replacement of the NVMM cache is based on the page granularity. This is because the replaced data should be written back to SSD before their eviction and the write unit of SSD is a logical file page. When a logical page is selected to be evicted from the NVMM cache, all the fragmented data associated with this logical page need to be written back to the SSD and then their space will be reclaimed for future caching. SFCM does not need to generate a new logical page by merging all the related dirty cachelines in the NVMM. Instead, the corresponding pages maintained in the DRAM are chosen to write to the SSD.

The main merit of SFCM is that it only writes the dirty cachelines of a synchronized page to the NVMM cache, which saves the wasted cache space caused by clean cachelines. However, when the dirty ratio of a synchronized page is close or equal to 100 percent, SFCM will degrade the cache utilization. This is because the fine-grained cache management introduces extra page metadata. More detailedly, every 4,096-bytes NVMM page has 512 bytes of metadata overhead. As a consequence, the saved cache space is not enough to amortize the introduced page metadata overhead with the SFCM scheme, thereby degrading the cache utilization and system performance.

3.2.2 Design of AFCM

To avoid the drawbacks of SFCM, we further propose an *Adaptive Fine-grained Cache Management* scheme. The key idea of AFCM is to adaptively choose fine-grained or coarse-grained cache management mode for each synchronized page according to its dirty ratio, thereby avoiding the drawback of the simple fine-grained approach.

The design of the AFCM scheme is shown in Fig. 5. In AFCM, the NVMM cache space is comprised of a few metadata pages and some dynamically allocated data pages. To meet the adaptive allocation requirement, AFCM contains two different types of data pages, which are fine-grained pages and coarse-grained pages respectively. The data organization of fine-grained pages is the same as that of NVMM pages in SFCM. On the contrary, the coarse-grained management is based on the page granularity. As shown in the figure, the coarse-grained pages do not contain any metadata, as the mapping table (i.e., the corresponding SSD addresses) of these coarse-grained pages are stored in the global metadata pages of the NVMM cache for crash recovery. Besides, the global metadata pages store 2-bits of state bitmap for each data page, which represents three states (i.e., *fine-grained*, *coarse-grained*, or *free* state). The version bitmap for coarse-grained data pages is also stored inside them.

The key challenge of AFCM is how to adaptively determine the cache mode (i.e., coarse-grained mode or fine-grained mode) for each synchronized page. To this end, we propose a *Cost-Benefit Model* based on the fine-grained cache management policy. In this model, we assume that the total number of all cachelines and dirty cachelines in a synchronized page are $Total\_Cnt$ and $Dirty\_Cnt$ respectively. Then, the saved cache space by the fine-grained cache mode is $(Total\_Cnt - Dirty\_Cnt) * 64$ bytes.[2] However, as analyzed in Section 3.2.1, each 4,096-bytes page only stores 56 cachelines of data, while the rest 512-bytes space is used for the page metadata, indicating that each dirty cacheline produces $512/56$ bytes of metadata overhead on average. As a result, the benefit and cost of the fine-grained cache mode can be represented by Equations (1) and 2 respectively.

$$\boldsymbol{Benefit} = (Total\_Cnt - Dirty\_Cnt) * 64 \tag{1}$$

$$\boldsymbol{Cost} = Dirty\_Cnt * (512/56). \tag{2}$$

Above model just ignores the metadata overhead caused by the coarse-grained mode, as it is almost negligible (less than 0.2 percent). Besides, the extra cost introduced by fine-grained indexing isn't taken into consideration either, since it is insignificant compared with the benefits of reduced I/O operations on SSD (typical latency of tens of microseconds). In AFCM, it chooses the fine-grained mode for a synchronized page only when the benefit is larger than the cost. Otherwise, it buffers this synchronized page using the coarse-grained mode. Worth noticing, if the currently synchronized page is determined to be buffered with coarse-grained mode, but was previously stored with fine-grained mode, AFCM needs to first merge all the related fragmented data in NVMM with the current synchronized DRAM page. Then, it writes the DRAM page to a newly-allocated coarse-grained NVMM page. Subsequently, it changes the state of this NVMM page from *free* to *coarse-grained* by modifying the state bitmap located in the global metadata page area. Finally, the NVMM spaces occupied by the original fragmented data of this logical page are released. The cache replacement is only different for the case where the currently synchronized page is coarse-grained and the to be evicted page is fine-grained. In AFCM, we choose the NVMM page that accommodates the largest part of the victim page for reclamation, and all the logical file pages corresponding to this victim will be evicted as well. As a result, more free space of NVMM is reclaimed.

The NVMM cache utilization of AFCM is much higher than that of SFCM especially when the average dirty ratio of all synchronized pages is close to 100 percent. Thus, AFCM shows better system performance than SFCM.

3.3 System Consistency and Crash Recovery

System consistency and crash recovery are the most important parts of the file systems. Leveraging the high performance and byte-addressability of NVMM, AFCM efficiently supports consistent file data update with *Transactional Copy-on-Write* mechanism. Besides, AFCM achieves fast crash recovery after a system crash by carefully organizing the metadata in NVMM.

2. In this paper, we assume that the CPU cacheline size is 64 bytes.

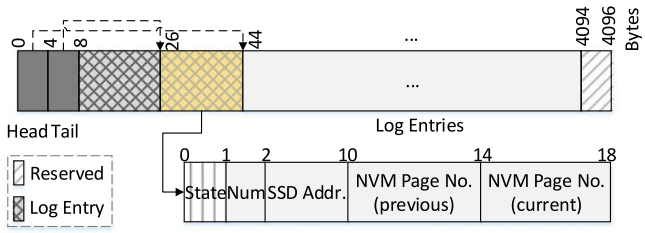


Fig. 6. The layout of circular log.

3.3.1 Transactional Copy-on-Write

Traditional file systems, as mentioned in Section 2.3, either fail to guarantee the data consistency (Ext2, etc.), leaving the file data partially updated during a system crash, or incur high overhead due to the double data write (Ext4 +JBD) or cascading updating (Btrfs). We achieve efficient and consistent file data update by providing atomic synchronization operations,[3] which are based on the transactional copy-on-write mechanism. With NVMM cache to take over the synchronized data flow, AFCM can provide consistent file data update by carefully managing the NVMM cache space, regardless of the implementation of underlayer file systems.

Transactional Copy-on-Write mechanism makes all data synchronized to NVMM out-of-place updated. For synchronization operation concerning multiple DRAM pages or buffered with fine-grained manner, the atomicity of it is guaranteed with a circular log, which is a 4 KB page placed at the tail of NVMM cache space (shown in Fig. 6).

For example, a synchronization operation containing two dirty data pages (denoted as P1 and P2) will be processed with the following steps:

1) Initialize the log entry pointed by *Head*, where the *Num*, *SSD Addr.* and *NVM Page No. (previous)* represent the number of dirty cachelines in P1, corresponding SSD address that stores this file page, and the previous NVM page index that caches P1 (empty if not hit in NVMM or previously cached with fine-grained mode).
2) Move *Head* forward by one in a circular manner.
3) If P1 is previously cached in NVMM, update the version bitmap (see Section 3) of P1 to `old` and update *State* to 1 (*State* will be checked to determine which phase has been finished during recovery).
4) Allocate new space for P1 according to the *Cost-Benefit Model*, update *NVM Page No. (current)* to point to the newly allocated space (empty when allocated with cacheline granularity), copy P1's data in it and update *State* to 2.
5) Repeat from step 1) to 4) to process P2 again.
6) Finally, release all the cache space of old version, update *Tail*to point to the same entry as that of *Head*.

For safe recovery, the modifications of the above six steps are strictly ordered with hardware instructions (e.g., `clflush` and `mfence`) to make data persistent. Besides, the

3. The term "atomic synchronization operation" is defined as: When system crashes during the execution of a `fsync` operation, the application sees either the previous or the current version of data consistently after rebooting.

atomicity guarantee of page replacement and reclamation during the synchronization operation also holds in TCoW. This is because the victim pages are released only after the page data has been written to the SSD device. Besides, we use 64-bits atomic operations provided by CPU to atomically reset the corresponding metadata when releasing the NVMM cache space. If the system crashes during this process, the primary data either lies in the NVMM cache or in the SSD device, or both, thus avoiding the system inconsistency issue.

Traditionally, those important applications (e.g., database systems) totally bypass the local file systems for data storage, which map a big file into the application and carefully update the data with self-defined transactional interfaces. With the consistent cache management in AFCM, the underneath file system naturally inherits the atomic synchronization semantic. To guarantee the consistency of application data, the only thing for upper-layer applications to do is correctly using the `fsync` operations. Our experiments show that the introduced consistency overhead by TCoW is restricted to 0.14 percent on average for all evaluated workloads.

3.3.2 Crash Recovery

As TCoW carefully updates the log entry, cached data and the corresponding metadata with strictly order, the design of crash recovery process is closely coupled with such order.

After rebooting from a system crash, the recovery process will first check the value of *Head* and *Tail* from the circular log. If they are not equal, the image of NVMM cache space needs to be rolled back as there are some unfinished transactions. Detailedly, the recovery process checks the *state* of each log entry from *Tail* to *Head* one by one. If all the log entries are at *State* of 2, It just scans the NVMM pages and releases any of them that are with old version of data. Otherwise, the cached data related to those log entries will be rolled back to the primary version with the help of information recorded both in log entries and NVMM metadata area: (1) discard all the newly allocated cache space, (2) reset the version bitmap of the related old version of data in NVMM cache, and (3) truncate the circular log by stepping back the *Head* pointer.

With above steps, the NVMM cache space is in a consistent state. Later, the state bitmap located in the global metadata page area of the NVMM cache is scanned in order to identify the state of each page (i.e., *fine-grained*, *coarse-grained*, or *free*) in the data page area. We then read the page mapping table in the global metadata area and the cacheline mapping table at the head of each fine-grained data page, to re-construct the NVMM Cache Index (described in Section 3.2.1). At the same time, the cacheline data in fine-grained pages and page data in coarse-grained pages are copied to DRAM cache to serve the normal read and write operations.

Note that some asynchronously written data may still be lost, as AFCM only guarantees the atomicity of synchronization operations. However, by correctly using such semantic, the applications can efficiently guarantee the consistency of stored data in AFCM without the bothering of maintaining their own log file. Besides, AFCM does not cache any file system metadata but only changes the I/O path of the file synchronization operations, thus all the file system metadata operations are processed in their original ways.

4 Implementation

In this section, we discuss the details of AFCM's implementation. As we use the trace-driven experiments to evaluate the AFCM scheme, we mainly develop a hybrid cache simulator based on DRAM and NVMM, and implement the AFCM scheme based on this cache simulator. In the hybrid cache simulator, the default page size of the DRAM cache is set to 4 KB. Moreover, we use the Least Recently Used (LRU) policy for the replacement of the DRAM cache. That is, the simulator maintains an LRU list to keep track of the recency of access references (including read and write references) of pages in the DRAM cache, and all the pages in the DRAM cache are sorted by their last access time. In contrast, the replacement of the NVMM cache is based on the Least Recently Written (LRW) algorithm. To implement this, the index list head of each logical page is linked to a global LRW list. When a cache extent of a logical page is updated, the index list head of this logical page would be moved to the Most Recently Written (MRW) position. While AFCM chooses LRU and LRW algorithms for the replacement of the DRAM and NVMM cache respectively, AFCM does not limit the DRAM or NVMM cache of using other sophisticated cache replacement policies. In fact, AFCM can support any cache replacement policies while still retaining their benefits of improving the cache hit ratio.

To use our simulator for the trace-driven experiments, we need to pre-process the trace files from various forms into a uniform format. This uniform format includes four different types of fields which are *file operation type* (i.e., read, write, fsync, or unlink), *file number*, *file access offset*, and *file access length*. In the process of replaying the traces, the cache simulator will first read each request record in the trace file, and then issue the corresponding request to the DRAM (i.e., read or write request) or NVMM cache (i.e., fsync request) based on the implemented AFCM scheme. When the free cache space in DRAM or NVMM is not enough, the cache simulator would clean enough cache space based on the corresponding replacement algorithms before the data is written to the cache.

5 Evaluation

In this section, we evaluate AFCM to address the following questions:

1) How does AFCM compare with existing cache management schemes in terms of performance? (see Section 5.2)
2) How does AFCM compare with existing cache management schemes in terms of flash lifetime and utilization of NVMM space? (see Sections 5.3 and 5.4)
3) How is AFCM sensitive to the NVMM and DRAM cache size? (see Section 5.5)
4) How does TCoW impact the overall performance? (see Section 6)

5.1 Experimental Setup

We use trace-driven experiments to evaluate AFCM, and choose six system-call traces (i.e., *Iphoto*, *Usr1*, *Usr2*, *Facebook*, *Twitter*, *TPCC*), which are collected in different environments. Specifically, *Iphoto* indicates the multimedia

TABLE 1
Description of Evaluated Cache Schemes

Name	DRAM granularity	NVMM granularity
DRAM-Cache	Page	N/A
SCCM	Page	Page
SFCM	Page	Cacheline
AFCM	Page	Page/Cacheline

application workload which is collected from the Mac OS X system using the DTrace tool by the University of Wisconsin, Madison [13]. The *Usr1* and *Usr2* traces are collected from the research desktop by FIU at different time respectively [34]. The *Facebook* and *Twitter* traces are obtained from an Android device when using the Facebook and Twitter apps [35]. Finally, the TPCC trace represents the database workload, which is collected from the DBT2 workload [36] running on the PostgreSQL 8.4.10 database system using the strace utility [37]. Moreover, we extract the read, write, fsync, and unlink operations from these system-call I/O traces and replay them on the cache simulator one by one.

All the experiments are conducted on an x86_64 Linux 4.2.3 kernel machine which is configured with 2.1 GHz Intel Xeon E5-2620 twelve-core processor and 16 GB physical memory. The operating system is Fedora 23. Moreover, the cache simulator is run on an LVM device consisting two commercial 128 GB SSDs named EJS1000 series from Jie Taike. In all experiments, we use DRAM to simulate the NVMM device. Unless otherwise specified, the default DRAM cache size is set to half of the workload size, and the default NVMM cache size is set to 20 percent of the workload size.

We implement two versions of the NVMM cache scheme, which are SFCM (see Section 3.2.1) and AFCM (see Section 3.2.2) based on the cache simulator. To compare with existing cache management policies, we also implement two conventional cache management schemes, named *DRAM-Cache* and *SCCM*. The description of above four cache schemes is shown in Table 1. *DRAM-Cache* represents the cache management scheme used in conventional file systems, in which the cache layer only contains the DRAM space and is managed in page granularity. In the *DRAM-Cache* scheme, the related data would be flushed to the SSD devices upon a synchronization operation. *SCCM* indicates conventional approach which manages the NVMM cache space in the coarse-grained manner. Moreover, it shares the same hybrid cache architecture as that of AFCM, but it manages the NVMM cache space in page granularity. For a fair comparison, we implement the same cache replacement algorithm in all versions of the cache management schemes. For all the experiments, each data-point is calculated using the average of at least 3 executions.

5.2 Overall Performance

We first evaluate the overall performance of all the cache schemes. Figs. 7a and 7b shows the execution time of replaying different traces normalized to that of DRAM-Cache.

From Fig. 7a we observe that AFCM outperforms DRAM-Cache by up to 8.8× (i.e., Usr2). The reason for this is that the DRAM cache cannot be used to buffer the synchronized data, indicating that all synchronization operation will generate the slow I/O operations to the SSD device, thereby reducing the system performance. When comparing AFCM with the conventional coarse-grained cache management scheme (i.e., SCCM), we observe that AFCM improves the system performance by up to 84 percent, because AFCM dynamically manages the NVMM cache in the fine-grained manner which improves the cache utilization of NVMM. Thus, AFCM can buffer more synchronized data in the NVMM cache compared to SCCM. While SFCM also uses the fine-grained approach to manage the NVMM cache, it underperforms SCCM in some cases (e.g., Facebook, Twitter, and TPCC). For example, in the case of TPCC workload, SFCM decreases the performance by 27 percent compared to SCCM. The reason behind this is that, when the dirty ratio of a synchronized page is close to 100 percent, the metadata space overhead of SFCM is larger than the saved space earned by the fine-grained cache management, thus providing worse cache utilization. We find that the average dirty ratio of all synchronized pages is larger than 99 percent for the TPCC workload. Therefore, SFCM decreases the system performance in this case. In contrast, AFCM always behaves the best in all the evaluated workloads, which demonstrates that the AFCM approach has better performance and wider usage than the SFCM one.

To verify the universality of AFCM and understand the overhead of cache replacement as described in Section 3.2.2, we evaluate the performance of above cache schemes with concurrent workloads, since they provide larger datasets. Four groups of workloads are chosen according to their in-page dirty ratio, which are ① both low dirty ratio, such as Usr1+Iphoto; ① both high dirty ratio, such as TPCC+Twitter; and ③ a hybrid combination, such as Facebook +Usr1+Usr2 and TPCC+Iphoto. As shown in Fig. 7b, unsurprisingly, the behavior of the four cache schemes has the same trend as that of Fig. 7a for concurrent workload groups

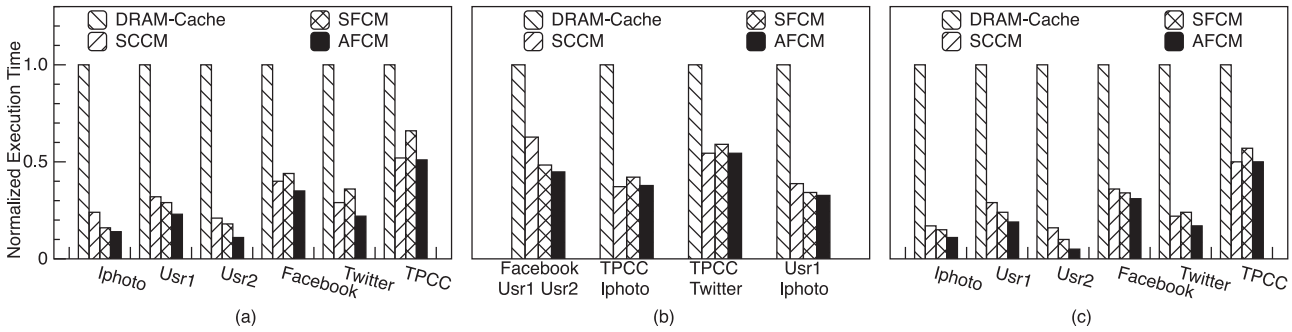


Fig. 7. (a) and (b) represents the overall performance of single and concurrent workload. (c) shows the overall flash lifetime of each workload.

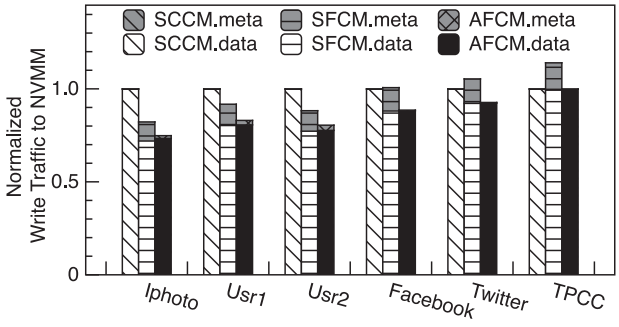


Fig. 8. Data and metadata traffic breakdown to NVMM.

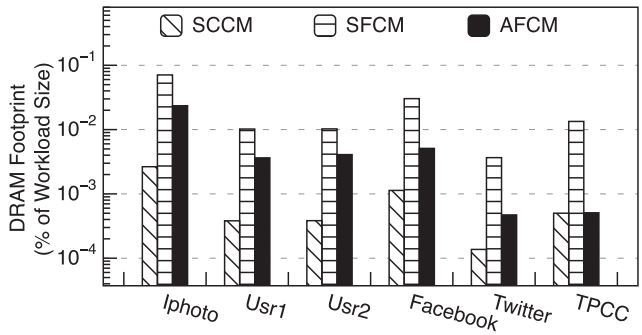


Fig. 9. DRAM footprint for file index.

in ① and ②. For hybrid workloads, we observe that AFCM still outperforms the other three systems. However, the winner between SFCM and SCCM varies in different workload groups, depending on which is the largest workload in each group. In conclusion, AFCM is capable of providing optimal performance regardless of the workloads behavior.

5.3 Flash Lifetime

To further investigate the endurance benefits of AFCM, we measure the SSD write pages of different cache policies respectively. The results are shown in Fig. 7c.

In this figure, we observe that the total write pages of DRAM-Cache are the most among all evaluated approaches, because it cannot buffer the synchronized data. To evaluate the impact of the cache utilization on the flash lifetime, we compare AFCM with SCCM. We can see that AFCM reduces the total write pages by 63 percent on average compared to SCCM, thereby extending the flash lifetime. This is because a higher cache utilization can lead to a less amount of data replacements, meaning that fewer data will be written back to the SSD device. In the case of Facebook workload, while SFCM reduces the SSD write pages by 5 percent compared to SCCM, its performance is still 6 percent lower than it. This is because SFCM increases the CPU overhead as the synchronized data need to be written to the NVMM cache based on the cacheline granularity rather than page granularity. We conclude that the coarse-grained cache management is much better than the fine-grained one when the dirty ratio of a synchronized page is close to 100 percent, because it leads to higher cache utilization and lower CPU overhead. As a result, AFCM is better than SFCM because it can adaptively choose the cache mode between the fine and coarse granularity.

5.4 Cache Space Utilization

NVMM Space Utilization. To understand the NVMM cache space utilization of different cache schemes microscopically, we collect their data and metadata traffic to NVMM by running all the workloads. The results are shown in Fig. 8 and we have the following observations: (1) SFCM shows the lowest Data Traffic among all the evaluated workloads, while SCCM presents the highest. This is because fine-grained cache mode in SFCM only writes dirty cachelines in synchronized pages, thus reducing the data traffic to NVMM. Actually, the data traffic of SFCM is the same as that of the total *Fsync Write* traffic for each workload. This also explains why the data traffic of the three cache schemes are always the same when the workloads are close to 100 percent in-page dirty ratio (e.g., TPCC). (2) SFCM presents the highest Metadata Traffic for all the workloads. As SFCM always caches the synchronized data with cacheline granularity, it has to store extra 512/56 bytes of metadata for each cacheline, leading to higher metadata traffic. (3) AFCM has the lowest Write Traffic (Data+Meta) for all the evaluated workloads, which reduces the write traffic by 13.4 and 10.6 percent on average when compared with SCCM and SFCM respectively. This is because AFCM dynamically chooses the cache mode depending on the dirty ratio of synchronized pages, so it keeps both data and metadata traffic at a relatively low standard. We conclude that AFCM shows the best behavior to reduce the write traffic to NVMM and extend its lifetime.

DRAM Space Usage. In this part, we collect the DRAM footprint of each cache scheme with all evaluated workloads, and the results are shown in Fig. 9. From the figure, we have the following observations: (1) SFCM and AFCM has higher DRAM footprints than that of SCCM, this is because both SFCM and AFCM need to maintain more complicated in-memory file indexes (e.g., linked-list) to track the cachelines of file data in the NVMM. (2) AFCM has lower memory footprint than SFCM, and the memory usage is controlled between 0.04 and 2 percent of the workload size for all evaluated workloads. As a whole, to exchange for the higher performance and reduced SSD write traffic, the extra memory consumption of AFCM is completely acceptable.

5.5 Sensitivity Analysis

Because the NVMM and DRAM cache size can affect the system performance and flash lifetime, we measure their impacts in this section.

5.5.1 Sensitive to the NVMM Cache Size

To evaluate the impact of the NVMM cache size on the system performance and flash lifetime, we choose two representative workloads (i.e., Iphoto and Usr2). In this set of experiments, we constantly set the DRAM cache size to half of the workload size, while varying the NVMM cache size from 5 to 20 percent of each workload size. Figs. 10 and 11 show the results for the Iphoto and Usr2 workload respectively.

In these two figures, we can see that the performance and endurance of DRAM-Cache remain the same with different NVMM cache sizes due to the absence of the NVMM cache in this scheme. Moreover, as the NVMM cache size becomes larger, the performance and endurance of other three schemes increase. This is because more synchronized data

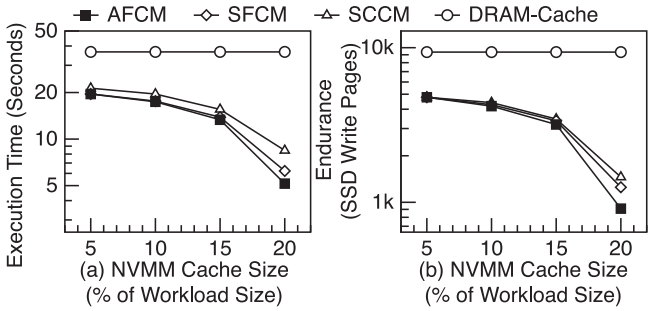


Fig. 10. Performance and flash lifetime with different NVMM cache sizes for the iphoto workload.

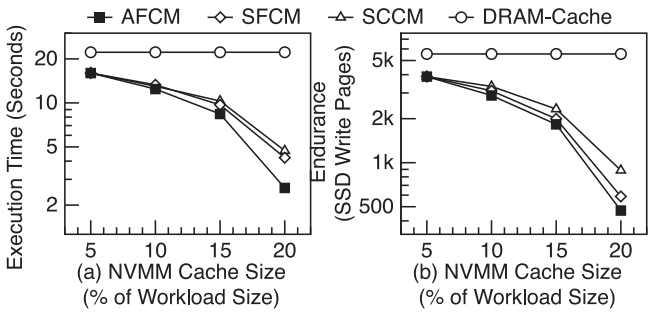


Fig. 11. Performance and flash lifetime with different NVMM cache sizes for the usr2 workload.

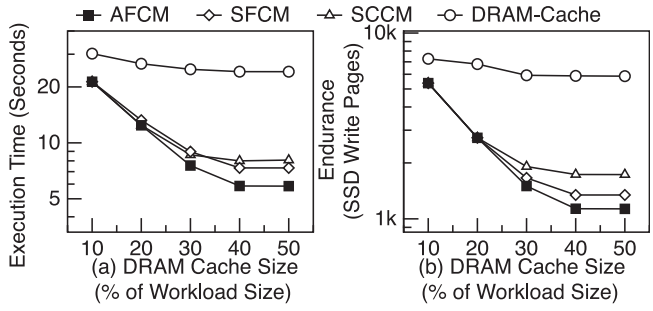


Fig. 12. Performance and flash lifetime with different DRAM cache sizes for the usr1 workload.

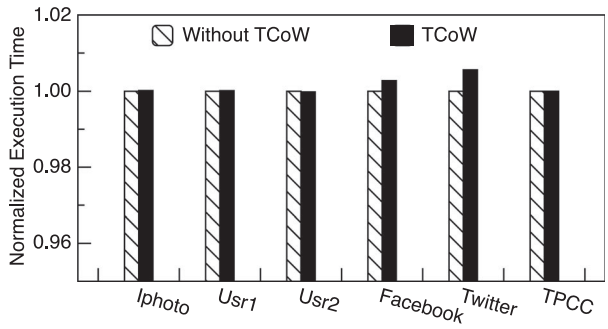


Fig. 13. Performance impact with TCoW.

can be buffered in the NVMM cache as the size of it increases, thereby increasing the cache hit ratio. A higher cache hit ratio can lead to better performance and less amount of data written back. In addition, we also make another observation from these two figures that the performance and endurance gap between AFCM and SCCM grows as the NVMM cache size increases. For example, for the Usr2 workload, AFCM improves the performance and endurance by 10 and 14 percent respectively compared to SCCM when the NVMM cache size is 10 percent of the workload size; In contrast, the performance and endurance of AFCM are increased by 84 percent and 2.19× respectively when the NVMM cache size is 20 percent of workload size. This is mainly due to that thrashing occurs if the NVMM cache is not large enough. Thus, the performance and endurance benefits due to the improvement of the cache utilization with AFCM cannot be fully exploited in this case. Even so, as the NVMM cache size becomes larger so that thrashing does not occur, AFCM will significantly increase the performance and endurance compared to conventional cache management schemes. In reality, the NVMM cache size should be configured based on the scale of the workloads. Our current conclusion is that AFCM shows significantly better performance and endurance within a certain range of the NVMM cache size.

5.5.2 Sensitive to the DRAM Cache Size

The DRAM cache size also has a strong impact on the system performance and flash endurance. To this end, we select Usr1 as a representative workload to measure the system performance and flash endurance with different DRAM cache sizes. In this part of experiments, the NVMM size is constantly set to 20 percent of each workload size. The results are shown in Fig. 12.

In this figure, we observe that the performance gap between AFCM and SCCM shrinks as the DRAM cache size decreases. For example, AFCM outperforms SCCM by 16 percent when the DRAM cache size is 30 percent of the workload size, while improving the performance by 36 percent over SCCM when the DRAM size is half of the workload size. This is because the DRAM cache replacement occurs too frequent when the DRAM size is small, thus a large amount of written data is forced to be flushed to the SSD device before the arrival of the synchronization operations. In such case, NVMM cache space is always not exhausted, so the improvement of the cache utilization is not helpful for improving the system performance. Moreover, we also make another observation from this figure that the DRAM cache size using DRAM-Cache approach has less impact on the performance and flash endurance than that using other three hybrid cache approaches. For example, as the DRAM cache size increased from 10 to 50 percent, the system performance only improved by 28 percent for the DRAM-Cache approach, while AFCM improves the system performance by about 3×. This is because the performance of DRAM-Cache is limited by the frequency of the synchronization operations in most cases, thereby, it has limited impact on the system performance. This observation demonstrates that the NVMM cache is crucial for improving the system performance and flash endurance.

5.5.3 Impact of Transactional Copy-on-Write

To guarantee the crash consistency of both NVMM cache space and file system image, AFCM introduces Transactional Copy-on-Write with a carefully managed circular log. To evaluate the impact of TCoW, for comparison, we implement a version of AFCM which removes the TCoW mechanism. The experimental results are shown in Fig. 13. From the figure, we find that AFCM with TCoW shows slightly

higher execution time in some of the evaluated workloads. This is because TCoW introduces extra write and persist operations to update the circular log. However, the overhead of TCoW is extremely low and almost negligible (0.14 percent on average).

6 Related Work

Cache Replacements. With the advent of emerging non-volatile main memory technologies, some researches have considered how to use small amounts of byte-addressable NVM to improve the performance and lifetime of flash-based SSDs. To improve the sequentiality of the buffer write-back operations, Flash-Aware Buffer (FAB) management [9] groups pages in the same flash block and evicts the group that has the largest number of pages when the buffer is full. However, FAB only considers the group size while overlooking the recency. To accommodate both the temporal locality and group size, the Cold and Largest Cluster (CLC) policy [8] combines the FAB and LRU algorithms.

Hybrid Architecture. UBJ [19] unions the function of page cache and journal device with NVMM space. The committed file data will be in-place updated in NVM, and later checkpointed to the disk to free the NVMM space. Tinca [20] consistently persists data into NVMM cache in a CoW way. It adopts role switch between journaling data and file data to avoid double writes. UBJ and Tinca have some semblance to AFCM in the architecture of cache management (NVMM +SSD/DISK), and Tinca also adopts a circular log to manage the cached data. However, both UBJ and Tinca organize the NVMM space with page granularity, and their designs are closely coupled in the architecture of Ext4, while the design of AFCM is not restricted in a particular file system, but benefits all the underneath file systems. Different from above systems, HiNFS is a NVMM-based file system designed to hide the high write latency of NVMM by caching the lazy-persistent write data in the DRAM cache, thus forming the DRAM-NVMM hybrid architecture. The data in DRAM cache is also managed with cacheline granularity, however, the crash consistency of the DRAM cache doesn't need to be considered.

Fine-grained Cache Managements. To exploit the byte-addressability of NVMM, PMFS [38] and Chen et al. [39] also adopt the fine-grained NVMM space management in the system designing. However, both of them target at the fine-grained journaling of metadata, because the updates of metadata tend to be small-sized. Unfortunately, all of existing works ignore the impact of the cache granularity on the system performance and flash lifetime for file data. As current NVMM technologies are still not mature and their capacity is limited, the coarse-grained (i.e., flash page [11] or flash block [8], [9], [10] granularity) cache management schemes used by existing buffer caching approaches significantly hurt the cache utilization and cache hit ratio, thereby reducing the system performance and flash endurance. As far as we know, AFCM is the first work to introduce an adaptive fine/coarse-grained cache management to manage the file data.

7 Conclusion

Frequent synchronization operations improve the system reliability but reduce the DRAM cache efficiency, resulting in poor system performance and flash endurance. In this paper, we observe that most of the written data need to be explicitly synchronized to the stable storage device in a wide range of workloads.

To resolve this problem, we propose to use the NVMM-based cache to reduce the synchronization traffic to the SSD device. To improve the space utilization of NVMM cache, we further propose an adaptive fine-grained cache management scheme called AFCM. It employs a hybrid cache architecture, which uses DRAM to serve the normal read and write operation without performance degradation, and let all the synchronized pages of data flow through NVMM cache space. To relieve the metadata overhead of fine-grained management, AFCM dynamically selects the fine-grained or coarse-grained mode to cache the data depending on its dirty ratio. A crash consistency mechanism named Transactional Copy-on-Write is also proposed to guarantee the consistency of both NVMM cache space and file system image. Our experimental results show that AFCM significantly improves the system performance and reduces the write traffic to SSD compared to the conventional coarse-grained cache management scheme.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61327902 and 61772300).

References

[1] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 181–192.

[2] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. 10th USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 139–154.

[3] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "EDM: An endurance-aware data migration scheme for load balancing in SSD storage clusters," in *Proc. 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 787–796.

[4] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 16:1–16:14.

[5] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: Integration of LRU and writes sequence reordering for flash memory," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, pp. 1215–1223, Aug. 2008.

[6] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2006, pp. 234–241.

[7] Z. Fan, D. H. C. Du, and D. Voigt, "H-ARC: A non-volatile memory based cache policy for solid state drives," in *Proc. 30th Symp. Mass Storage Syst. Technol.*, Jun. 2014, pp. 1–11.

[8] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Trans. Comput.*, vol. 58, no. 6, pp. 744–758, Jun. 2009.

[9] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-aware buffer management policy for portable media players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.

[10] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based SSDs," *ACM Trans. Storage*, vol. 8, no. 1, pp. 1:1–1:24, Feb. 2012.

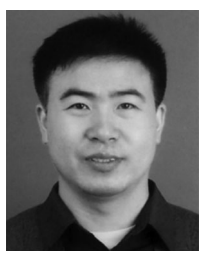
[11] Q. Wei, C. Chen, and J. Yang, "CBM: A cooperative buffer management for SSD," in *Proc. 30th Symp. Mass Storage Syst. Technol.*, Jun. 2014, pp. 1–12.

[12] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the sync," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 1–14.
[13] 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 *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 71–83.
[14] Atomic commit in sqlite, (2018). [Online]. Available: https://www.sqlite.org/atomiccommit.html
[15] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *Trans. Storage*, vol. 8, no. 4, pp. 14:1–14:25, Dec. 2012.
[16] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating periodic flush overhead of file I/O with non-volatile buffer cache," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1145–1157, Apr. 2016.
[17] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.
[18] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 14–23.
[19] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.
[20] Q. Wei, C. Wang, C. Chen, Y. Yang, J. Yang, and M. Xue, "Transactional NVM cache with high performance and crash consistency," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 56.
[21] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, 1973, vol. 973.
[22] P. J. Denning,"The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, 1968.
[23] D. Willick, D. Eager, and R. Bunt, "Disk cache replacement policies for network fileservers," in *Proc. 13th Int. Conf. Distrib. Comput. Syst.*, May 1993, pp. 2–11.
[24] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 439–450.
[25] M. Cao, S. Bhattacharya, and T. Tso, "Ext4: The next generation of ext2/3 filesystem," in *Proc. Linux Storage Filesystem Workshop*, 2007, Art. no. 1.
[26] C. Mason, "The btrfs filesystem," (2007). [Online]. Available: oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf
[27] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[28] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
[29] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 228–243.
[30] R. Hagmann,"Reimplementing the Cedar file system using logging and group commit," *ACM SIGOPS Operating Syst. Rev.*, vol. 21, no. 5, pp. 155–162, 1987.
[31] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1996, pp. 1–1.
[32] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *Proc. USENIX Winter Tech. Conf.*, 1994, pp. 19–19.
[33] E. Kustarz, "ZFS-the last word in file systems," 2008. [Online]. Available: http://www.opensolaris.org/os/community/zfs/
[34] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche, "Non-blocking writes to files," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 151–165.
[35] E. Laboratory, "Mobibench benchmark tool," 2013. [Online]. Available: http://www.mobibench.co.kr/
[36] Dbt2 test suite, (2015). [Online]. Available: http://osdldbt.sourceforge.net/
[37] strace(1) - linux man page, (1995). [Online]. Available: http://linux.die.net/man/1/strace
[38] S. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 15.
[39] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proc. 32nd Symp. Mass Storage Syst. Technol.*, 2016, pp. 1–13.

Youmin Chen received the BS degree in computer science from Beihang University, in 2016. He is currently working toward the second year PhD degree in the Department of Computer Science and Technology, Tsinghua. His current research interests include non-volatile memories, distributed systems and file systems.

Youyou Lu received the BS degree in computer science from Nanjing University, in 2009, and the PhD degree in computer science from Tsinghua University, in 2015. He is an assistant professor with the Department of Computer Science and Technology, Tsinghua. His current research interests include non-volatile memories and file systems. He is a member of the IEEE.

Pei Chen is working toward the master degree in the Department of Computer Science and Technology, Tsinghua. His current research interests include non-volatile memories and file systems.



Jiwu Shu received the PhD degree in computer science from Nanjing University, in 1998, and finished the postdoctoral position research at Tsinghua University, in 2000. He is a professor with the Department of Computer Science and Technology, Tsinghua University. His current research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing. He has been teaching at Tsinghua University. He is an IEEE fellow.