TESA: a Temporal and Spatial Information Aware Writeback Policy for Home Network-Attached Storage Devices

Ting-Chang Huang and Da-Wei Chang, *Member*, IEEE

Abstract — *Home Network-Attached Storage (NAS) provides an easy way for data sharing and backup among multiple consumer electronic devices in home networks. Because of large capacity and cost effectiveness, disks are widely adopted in home NAS devices. In addition, data writes are common for many home NAS devices since these devices are usually used for data storage and backup.*

A writeback policy selects which dirty buffers are to be flushed to the disk, which is critical to the system performance under write-intensive workloads. In this paper, an intelligent writeback policy for home NAS called TESA is proposed. In contrast to most existing writeback policies, TESA considers both temporal and spatial information of the dirty buffers to improve the writeback performance. Considering the temporal information helps to reduce the frequency of writing back recently used dirty buffers, and hence leads to reduced write traffic. Considering the spatial information causes a reduction in the overall seek time and rotation delay of the dirty buffer writeback.

The TESA writeback policy was implemented on a NAS evaluation board running Linux kernel. The performance results shows that TESA yields a significant performance improvement (i.e., up to 33.1%) over the original writeback policy of Linux. Moreover, TESA outperforms the policy that considers only the temporal information of the dirty buffers by up to 21.0%, and it also has up to 10.9% performance improvement over a previous policy that considers both the temporal and spatial information[1].

Index Terms —Writeback policy, Buffer management, Home network-attached storage, Temporal information, Spatial information.

I. Introduction

Nowadays, data sharing among multiple consumer electronic devices in a home network is easy [1], and the range of sharing can also be beyond a single home network [2], [3]. Home Network-Attached Storage (NAS) provides a large storage space for data sharing among multiple consumer electronic devices in one or more home networks [4]. For example, users can create or download media content (e.g. video files) with video game consoles or set-top boxes (STBs), and store the content in a home NAS device. The stored content can then be accessed by the other devices (e.g. TVs or smart phones). In addition, home NAS is often used to backup data from consumer electronic devices in the home networks. For example, users can backup vital data from their personal mobile devices (e.g. laptops or smart phones) to a home NAS device. Users can also backup video content captured by camcorders (i.e., video camera recorders) to a home NAS device. Fig. 1 shows the role of home NAS.

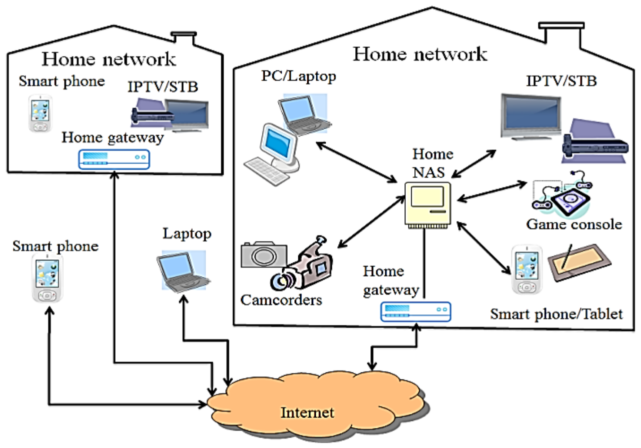


Fig. 1. Role of home NAS in home networks

Currently, embedded operating systems (e.g. embedded Linux) are widely adopted in home NAS devices. In addition, disk is a common storage medium in a home NAS device because of its large capacity and cost effectiveness. However, due to the large performance gap between disks and processors, disks are the major performance bottleneck in home NAS devices. Moreover, owing to small performance improvement of the mechanical accesses in disks such as plate rotation and disk head seeking, the gap is becoming larger. According to a previous study, processor speed grows 60% annually but the annual performance improvement of disks is only 8% [5]. Therefore, for performance consideration, operating systems use memory as buffer caches to buffer writes to the disks and to cache recently-read data.

[1] This work was supported in part by the National Science Council, Taiwan, Republic of China, under Grant No. NSC 101-2221-E-006-098-MY3.

T. C. Huang is with the Department of Computer Science, National Chiao Tung University, No. 1001, University Road, Hsinchu, Taiwan 300, R.O.C. (e-mail: tchuang@cs.nctu.edu.tw).

D. W. Chang is with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan, Taiwan 701, R.O.C. (e-mail: davidchang@csie.ncku.edu.tw).

Contributed Paper
Manuscript received 12/26/12
Current version published 03/25/13
Electronic version published 03/25/13.

Disk writes are common for many home NAS devices since home NAS is usually used for data storage and backup. Using memory to buffer disk writes can effectively improve the response time in these devices. However, since main memory is volatile, dirty buffers (i.e., buffers with data more up-to-date than their counterparts in the disk) still need to be written back to the disks to reduce the loss of up-to-date data upon sudden power outages or system crashes.

To prevent losing a large amount of up-to-date data upon power or system failures, writeback of dirty buffers is performed when the amount of dirty buffers reaches a specific threshold. The writeback threads of the operating system are responsible for performing the writeback task. Specifically, these threads select target dirty buffers according to the *writeback policy*, and then flush these buffers to the disk.

The writeback policy has a significant impact to the system performance when there are a large number of dirty buffers in the memory and the free memory drops below a threshold (i.e., memory pressure). Write-intensive workloads with large working sets could easily lead to such condition. Under this condition, the system performance degrades severely since I/O operations (which require free memory space either as the write buffers or as the read caches) have to be blocked until enough dirty buffers have been written back and the corresponding memory space has been reclaimed [6]. Therefore, a writeback policy should select dirty buffers that can be efficiently written back to the disk in order to reduce the performance degradation.

To efficiently write back dirty buffers, the disk block locations (i.e., spatial information) of the dirty buffers should be considered to reduce the seek time and the rotation delay. However, such information is not considered by the writeback policies of existing operating systems. For example, the Linux operating system uses a file based policy, which writes back dirty buffers in a file-by-file manner, and some operating systems write back dirty buffers that are not used recently.

In this paper, an efficient writeback policy for home NAS called TESA is proposed. TESA improves the writeback performance by exploiting both the spatial and temporal information of the dirty buffers. Similar to the writeback policies of some existing operating systems, TESA considers the temporal information of the dirty buffers. Specifically, it reduces the frequency of writing back recently-used dirty buffers since these buffers are likely to become dirty again soon after they have been written back. More importantly, TESA considers the spatial information of the dirty buffers to reduce the seek time and the rotation delay. Two types of spatial information are considered in the design of TESA, and the corresponding performance is shown in this paper.

The TESA writeback policy was implemented on a NAS evaluation board running Linux 2.6.12, and 6 widely-used benchmarks were used to evaluate the performance of TESA. The performance results show that TESA outperforms the writeback policy used in Linux by up to 33.1%, the policy that considers only the temporal information by up to 21.0%, and a previous policy that considers both the temporal and spatial information (similar to DULO [7]) by up to 10.9%.

The remainder of this paper is organized as follows. Section II describes the related work, followed by the design and implementation of TESA in Section III. Section IV shows the performance results, and conclusions are given in Section V.

II. Related Work

Traditional operating systems flush dirty buffers without considering the temporal or spatial information of the buffers. For example, the Linux operating system flushes dirty buffers in a file-by-file manner. That is, dirty buffers are grouped by files, and the dirty buffers belonging to a file are flushed before the flushing of the dirty buffers belonging to another file. However, one problem of this policy is that it may flush recently-used dirty buffers since temporal information is not considered. Writing back these buffers does not help relieving memory pressure since they tend to be kept in memory by LRU based page replacement algorithms, which are commonly used in operating systems. Moreover, these buffers are likely to become dirty again soon after they have been flushed, and therefore they require to be flushed later. This increases the write traffic to the disk. Some operating systems avoid this problem by only writing back dirty buffers that are not used recently. However, all of the above policies do not consider the spatial information of the dirty buffers, and thus could lead to longer seek time and rotation delay during the writeback. The proposed writeback policy TESA considers both temporal and spatial information of the dirty buffers, resulting in superior performance.

DULO [7] and WOW [8] also exploit both temporal and spatial information of the dirty buffers. DULO groups dirty buffers corresponding to adjacent blocks as a segment and flushes the largest segment among the non-recently-used buffers. If a fixed amount of dirty buffers need to be flushed, flushing larger segments leads to a reduced number of flushed segments and hence results in less total seek time and rotational delay. However, under workloads with random and small writes, segments tend to be small and hence the benefit of DULO is limited [9]. WOW is a writeback policy used in a storage controller. It divides the storage into a set of contiguous blocks, called write groups, locates write groups that are not written recently in the CSCAN order, and flushes the dirty buffers in the located write groups. However, most operating systems already adopt SCAN-like I/O schedulers [10] and therefore flushing dirty write groups in the CSCAN order does not have much benefit. Moreover, WOW may flush a large number of small write groups (i.e., write groups with small numbers of dirty buffers) to relieve memory pressure since the amount of dirty buffers in each write group is not considered. For example, if the operating system requires reclaiming 100 dirty buffers, WOW may flush 10 write groups, whereas flushing the largest write group (which may have more than 100 dirty buffers) may be enough to relieve the memory pressure. Therefore, WOW may lead to more disk positioning time under such condition.

AWOL [6] improves the performance of writing back dirty buffers by adjusting the start/stop time of dirty buffer flushing according to the workload. TESA takes a different approach to improve performance of dirty buffers flush (i.e., by considering the temporal and spatial information of the dirty buffers).

Efficient disk scheduling algorithms, like SPTF [10] or CSCAN, also help to reduce the latency of flushing dirty buffers. The difference between a writeback policy and a disk scheduling algorithm is that the former generates write requests, which are scheduled by the latter. Both writeback policies and disk scheduling algorithms play important roles in latency reduction of flushing dirty buffers.

III. Design and Implementation

As mentioned above, a writeback policy (which is responsible for selecting target buffers to be flushed) is critical to the writeback performance. In this paper, the TESA writeback policy is proposed to improve the writeback performance by exploiting both the spatial and temporal information of the dirty buffers. By exploiting the spatial information of the dirty buffers, TESA reduces the seek time and rotation delay when flushing dirty buffers, leading to higher flushing speed. Moreover, by exploiting the temporal information, TESA reduces the frequency of flushing frequently modified buffers. As mentioned in Section II, writing back such buffers increases the write traffic to the disk and does not help to relieve memory pressure.

In the following, the consideration of spatial and temporal information of dirty buffers in TESA is first described, followed by the description of the implementation of TESA in Linux.

A. Exploiting Spatial Information

Two schemes exploiting spatial information, the zone scheme and the zone_segment scheme, are proposed in TESA to improve the speed of flushing dirty buffers. The zone scheme selects dirty buffers that are close to each other in the block locations as the targets. This reduces the seek time by decreasing the seek distance. According to the previous study, seek time is reduced with the decrease of the seek distance [11]. Therefore, the performance of flushing dirty buffers can be improved if the seek distances are decreased. To achieve this, the disk is divided into a number of fixed-size zones, each of which corresponds to a set of contiguous blocks on the disk, and the *zone* scheme selects the zone with the largest number of dirty buffers (i.e., the maximum zone) as the target zone and flushes the dirty buffers belonging to the target zone. The set of the maximum zones $Zones_{Max}$ is computed by

$$Zones_{Max} = \{Z_i \in SZ \mid \forall Z_j \in SZ,\ Z_j \neq Z_i,\ NB_j \leq NB_i\} \quad (1)$$

where SZ denotes the set of all the zones that have dirty buffers and NB_i denotes the number of dirty buffers in zone i. If there are multiple maximum zones in the $Zones_{Max}$, the zone with the minimum zone identifier in the $Zones_{Max}$ is selected as the target zone. If the number of buffers that need to be flushed is larger than the number of buffers in the target zone, the next maximum zone is selected again according to (1) and the steps repeat until enough buffers have been flushed.

Note, since the operating system does not know the physical location of a given block, the logical block number (LBN) is used as an estimation of the physical block location. For example, given the zone size as 100 blocks, blocks with LBN 10 and LBN 20 both belong to zone 0. In this way, it is assumed that blocks having close LBNs are also close in the physical block locations. Although a modern disk hides the physical location of a block and presents only the LBN of that block to the host operating system, this assumption generally holds and is commonly used [12], [13].

Although reducing long seeks, the *zone* scheme may still incur a significant number of short seeks (companied with rotation delays) when flushing dirty buffers. For example, flushing block 20 right after the writeback of block 10 still requires a short seek and a following rotation delay. To address this, the *zone_segment* scheme tries to flush a zone containing long segments. A segment is a set of dirty buffers that correspond to a set of adjacent blocks on the disk. Flushing a whole segment typically requires only a single seek and a following rotation delay. As a result, the *zone_segment* scheme reduces not only the seek distance between two successive buffer flushes but also the number of seeks and rotation delay.

To locate the zones with long segments, the average segment length for each zone i, denoted as ASL_i, is maintained in the *zone_segment* scheme. The ASL_i can be calculated by

$$ASL_i = \begin{cases} NB_i/NS_i, & if\ NB_i \geq NB_{avg} \\ 0 \quad , & otherwise \end{cases} \quad (2)$$

where NB_i and NS_i denote the number of dirty buffers and the number of segments in zone i, respectively, and NB_{avg} denotes the average number of dirty buffers in all zones. Note that a zone with a large average segment length may have only a small number of dirty buffers. Thus, selecting target zones purely based on the average segment length of each zone could result in flushing a large number of zones that contain small numbers of dirty buffers, which would lead to poor writeback performance if the distances among these zones are long. To prevent this, in (2), the ASL value is set as 0 if the number of dirty buffers in a given zone is less than a threshold. Currently, the threshold is set as the average number of dirty buffers per zone. The set of zones with the maximum ASL value, denoted as $Zones_{MaxASL}$, is computed by

$$Zones_{MaxASL} = \{Z_i \in SZ \mid \forall Z_j \in SZ, Z_j \neq Z_i, ASL_j \leq ASL_i\} \quad (3)$$

Note that flushing zones with the maximum ASL value can achieve good writeback performance only when the maximum ASL value is large. Under that situation, long segments can be flushed. However, under workloads with small random writes, all the segments could be short and the maximum ASL value could be small. Flushing zones with the maximum ASL value has little benefit for these workloads. Under these workloads, the zone with the largest number of dirty buffers should be flushed in order to reduce the inter-zone seeks. Therefore, the *zone_segment* scheme selects the target zones that need to be flushed according to (4).

$$Zones_{target} = \begin{cases} Zones_{MaxASL}, & if\ ASL_{Max} \geq \alpha \\ Zones_{Max}, & otherwise \end{cases} \quad (4)$$

In (4), the target zones are selected from $Zones_{Max}$ if the maximum *ASL* value ASL_{Max} is smaller than a pre-defined threshold *α*. Otherwise, the target zones are selected from $Zones_{MaxASL}$. If the target zones are selected from $Zones_{MaxASL}$ and there are multiple zones in $Zones_{MaxASL}$, the zone with the maximum dirty buffers is selected as the target zone. The dirty buffers in the target zone are flushed. If the number of buffers that need to be flushed is larger than the number of buffers in the target zone, the next target zone is selected again according to (4), and the steps repeat until enough buffers have been flushed.

Fig. 2 illustrates the target zone selection in the *zone_segment* scheme. In Fig. 2, each zone consists of 25 blocks, the number shown on each block indicates the logical block number (LBN), and *α* is set as 2. Initially, the value of NB_{avg} is assumed to be 10.3. The *ASL* values of both zone 3 and zone 5 are zero since the numbers of dirty buffers in these zones are both smaller than NB_{avg}. Zone 4 is selected as the target zone since it has the largest *ASL* value 10, which is larger than *α*. At the bottom of the figure, the dirty buffers of zone 4 have been flushed and the NB_{avg} is assumed to become 7. As can be seen, although zone 5 now has the maximum *ASL* value among the zones, its *ASL* value (i.e., 1.4) is smaller than *α*. Therefore, zone 2 (i.e., the zone with the maximum number of dirty buffers) will be selected as the target zone if more dirty buffers need to be flushed.

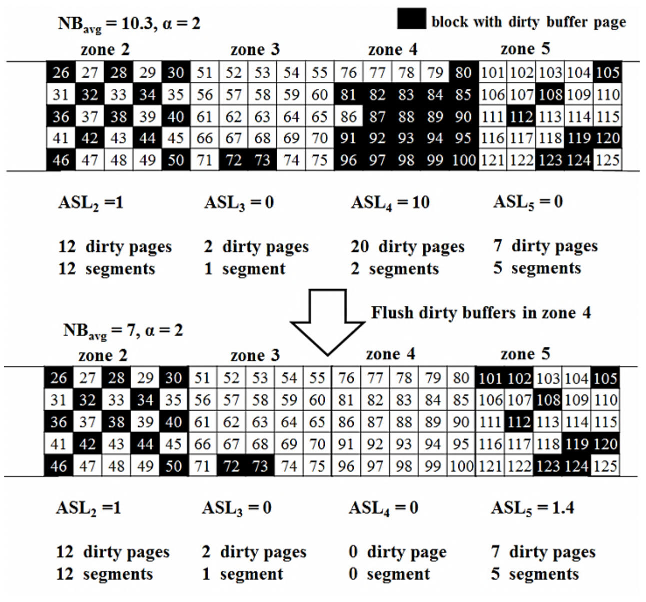


Fig. 2. Target zone selection in the zone_segment scheme

B. Exploiting Temporal Information

As mentioned above, flushing frequently modified buffers does not help to relieve memory pressure since these buffers tend to be kept in the memory by the memory management subsystem. Moreover, flushing these buffers increases the disk write traffic since these buffers are likely to become dirty again soon after they have been flushed. To address this problem, in addition to exploiting the spatial information of the buffers, TESA also takes the temporal information into account.

Many modern operating systems keep track of temporal information for each buffer page, and hence TESA can determine whether or not a given buffer is recently-used based on that information. For example, an operating system may maintain an LRU list for page replacement. In that case, the TESA writeback policy can prevent writing back a dirty buffer that is close to the MRU (Most Recently Used) end of the list. For another example, Linux maintains a flag called *PG_active* for each page, which is set as 1 if the page is recently-used. Therefore, in Linux, TESA can exclude all the buffers with the *PG_active* flags set as 1 when flushing buffers.

Note that the frequently modified dirty buffers would still be flushed eventually even they have been excluded by TESA. In order to prevent dirty buffers from being kept in the memory for a long time, many operating systems periodically write back old dirty buffers (i.e., dirty buffers that have not been flushed for more than a specific time period). When TESA is used in such operating systems, the frequently modified buffers can be flushed via this periodic writeback procedure.

C. Implementation of TESA

To exploit both spatial and temporal information of the dirty buffers, the *zone* and *zone_segment* schemes of TESA are modified to exclude frequently modified buffers. Fig. 3 shows the data structures used by TESA to maintain dirty buffers. As shown in Fig. 3, dirty buffers corresponding to the same zone are grouped together via a zone control block (*zone_cb*). Each *zone_cb* structure includes the information such as the zone number and the number of dirty buffers in that zone. In the *zone_segment* scheme, extra information is included in the *zone_cb* structure such as the number of segments in that zone and a block bitmap. Each bit in the block bitmap corresponds to a block in that zone and indicates whether there is a dirty buffer corresponding to the block. Moreover, dirty buffers belonging to the same segment are grouped in a data structure called segment control block (*seg_cb*). As shown in Fig. 3(b), zone 1 has 2 segments, one contains dirty buffer pages corresponding to blocks 256 to 305 (i.e., 50 blocks), and the other contains dirty buffer pages corresponding to blocks 512 to 611 (i.e., 100 blocks). To speed up the search of a specific segment, segments in each zone are managed by a hash table of 256 entries. Each entry chains a list of segments and a segment starting with LBN *B* is chained in the list corresponding to the entry (*B* mod 256).

When a dirty buffer needs to be inserted into the data structures shown in Fig. 3, the corresponding *zone_cb* structure has to be located first. This is achieved by calculating the zone number according to the LBN of the buffer and then searching the *zone_list*. In the *zone* scheme, the dirty buffer can then be chained directly under the located *zone_cb* structure. In the *zone_segment* scheme, however, the corresponding *seg_cb* structure has to be located. As described above, the starting LBN of a segment is used as the hash key to locate a specific *seg_cb* structure. Thus, to locate the *seg_cb* structure corresponding to the dirty buffer, the block bitmap in the *zone_cb* structure is checked to find the starting LBN of the segment. Note that inserting a buffer may involve creating a new segment, expanding an existing segment, or merging segments. For example, assuming that two segments are presented, segment *S1* corresponds to LBNs 1 to 3 and segment *S2* corresponds to LBNs 5 to 8, and a dirty buffer with LBN 4 needs to be inserted. By checking the block bitmap, the dirty buffer with LBN 4 can be chained into the *seg_cb* structure of *S1*, which can be located by using LBN 1 as the hash key. In addition, all the dirty buffers in *S2* need to be moved to the *seg_cb* structure of *S1*.

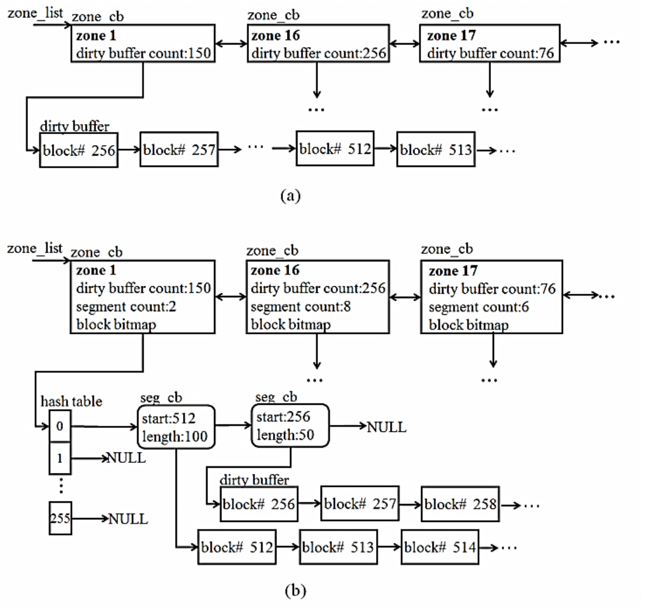


Fig. 3. Data structures used in the (a) zone and (b) zone_segment schemes

In the *zone_segment* scheme, a segment can span across zone boundaries, and a segment is assigned to a zone if the starting LBN of the segment belongs to that zone. The *seg_cb* structure of a segment *S* is placed in the hash table of a *zone_cb* structure of a zone *Z* if *S* is assigned to *Z*. For example, if each zone contains 3 blocks, the segment *S1* in the above example spans across the boundary of zone 0 and zone 1, and the *seg_cb* structure of *S1* is placed in the hash table of the *zone_cb* structure of zone 0 (i.e., *S1* is assigned to zone 0). As a consequence, inserting a dirty buffer (which requires locating the *seg_cb* structure corresponding to the dirty buffer) may need to check the block bitmaps of multiple zones. As in the above example, if each zone contains 3 blocks, inserting dirty buffer with LBN 4 requires checking the block bitmaps of both zone 1 and zone 0 to obtain the starting LBN of the segment *S1*.

In the implementation, both the *zone_cb* and the *seg_cb* structures are allocated and freed dynamically. For example, if the last dirty buffer belonging to a zone/segment is disassociated from the zone/segment (i.e., the buffer becomes clean or has been freed), the corresponding *zone_cb*/*seg_cb* structure is freed.

The TESA writeback policy was implemented in the Linux kernel 2.6.12. Specifically, the following modifications were made. First, the data structures shown in Fig. 3 were implemented. In Linux, a buffer is represented by a *buffer_head* structure. The *buffer_head* structure was augmented to include a field for chaining the dirty buffers in the list shown in Fig. 3.

Second, the functions that change the *BH_Dirty* flag or the *PG_active* flag of a buffer were modified to associate/disassociate the buffer to/from its corresponding *zone_cb* and *seg_cb* structures. A buffer is dirty if its *BH_Dirty* flag is set. TESA only associates a buffer with the above structures when the buffer is dirty since it only needs to flush dirty buffers. In addition, as mentioned above, TESA tries to avoid flushing frequently modified buffers. To achieve this, only infrequently-used dirty buffers (i.e., dirty buffers with *PG_active* flag set as 0) were placed in the structures shown in Fig. 3. Therefore, when the *PG_active* flag of a dirty buffer is set as 1 (for example, during the invocation of the function *add_page_to_active_list()*), the buffer is removed from the corresponding *zone_cb* and *seg_cb* structures. Similarly, when the *PG_active* flag of a dirty buffer is set as 0 (for example, during the invocation of the function *del_page_from_active_list()*), the buffer is placed into the corresponding *zone_cb* and *seg_cb* structures.

Third, the function *background_writeout()* was modified, which is invoked by the dirty buffer flushing threads (i.e., the *pdflush* thread) to write back dirty buffers when the amount of dirty buffers exceeds a specific threshold. Originally, this function utilizes the file-based writeback policy, under which the dirty buffers belonging to a file are flushed before the flushing of the dirty buffers belonging to another file. The file-based writeback policy was replaced with the *zone* and *zone_segment* schemes of TESA. Note that the function that is invoked periodically to flush old dirty buffers (i.e., the *wb_kupdate()* function) was kept intact so as to allow the data updates to be flushed to the disk within a predefined time limit, as in the original Linux.

IV. Performance Evaluation

A. Experimental Environment and Workloads

The performance of the TESA writeback policy was evaluated on a NAS evaluation board equipped with a 1.6 GHz processor, 1GB RAM and a 500GB disk (7200 RPM). Linux (kernel version 2.6.12) was run on the board and ext3 was used as the file system. The following 6 benchmarks were used for performance evaluation: *seq-write*, *rnd-write*, *postmark*, *fileserver*, *videoserver*, and *data-backup*.

The *seq-write* and *rnd-write* benchmarks are included in the *filebench* file system benchmark suite. The *seq-write* benchmark creates a single empty file first and then performs a sequence of append operation on this file. The data size for each append operation is 4KB. The *rnd-write* benchmark repeatedly issues 16KB random writes to a 2GB file. The *fileserver* benchmark (included in *filebench*) simulates the workload of a file server, which performs a sequence of creation, deletion, append, read, and write operations. The initial file set contains 10k files with mean size 128KB. The sizes of each read/write operation and each append operation are 1MB and 16KB, respectively. The *videoserver* benchmark (included in *filebench*) simulates the workload of a video server, in which 4 1GB video files are created and then read concurrently by 4 threads. The run time of all the above benchmarks are set to 60 seconds, the default value of the benchmarks. *Postmark* [14] is a widely used benchmark for evaluating storage system performance. During the execution, it creates an initial set of files and then applies a number of transactions on those files. Each transaction consists of a create/delete operation together with a read/append operation. The initial file set contains 100K files residing in 100 directories, and 200K transactions are performed. The *data-backup* benchmark simulates user data backup from a laptop to the NAS. The backup data includes 18630 files and the total size is 649MB.

In an operating system, dirty buffers can be flushed under different conditions. In the following, the percentages in the amount of dirty buffers flushed under different conditions in Linux are presented first. Next, the performance of TESA under different zone sizes and α values are shown, which is followed by the performance comparison of the *zone* and the *zone_segment* schemes. Finally, the performance of TESA is compared with that of several existing writeback policies, and the memory overhead of TESA is evaluated.

B. Amount of Dirty Buffers Flushed under Different Conditions

As mentioned in Section III-C, TESA was implemented by modifying the *background_writeout()* function in Linux, which is invoked in the condition where the amount of dirty buffers exceeds a specific threshold. Generally, there are three other conditions for writing back dirty buffers. First, writeback is performed periodically to prevent dirty buffers from being kept in the memory for a long time. Second, dirty buffers are written back by the memory reclamation procedure for obtaining more free memory upon memory pressure. Third, writeback is performed by the checkpointing procedure of a journaling file system (e.g. ext3) to reclaim its journal space [15]. The amount of dirty buffers flushed under different conditions in Linux was measured. According to the results, almost all the dirty buffers (i.e., more than 97.6%) are written back due to that the amount of dirty buffers exceeds a specific threshold. Therefore, TESA can simply be implemented in the *background_writeout()* function without noticeable reduction in its effectiveness.

C. Performance Impact of Zone Sizes and α Values

Both the *zone* scheme and the *zone_segment* scheme flush dirty buffers in a zone-by-zone manner, and the zone size has performance impact to these schemes. For example, in the *zone* scheme, extremely large zones could lead to long seeks within a zone when the zone is flushed, and extremely small zones result in the flush of many (small) zones and could lead to long seeks among the flushed zones. Fig. 4 and Fig. 5 show the performance results of the *zone* and the *zone_segment* schemes, respectively, under various zone sizes. The results are normalized to the performance with 2MB zones under the given scheme. For the *seq-write*, *rnd-write*, *fileserver* and *videoserver* benchmarks, the performance was measured in terms of the number of file operations performed per second. Therefore, higher performance means a larger number of file operations performed per second. For the other benchmarks, the performance was measured in terms of execution time. Therefore, higher performance means shorter execution time. According to the figures, setting the zone size as 8MB allows both the *zone* and the *zone_segment* schemes to achieve good performance under all the workloads. Thus, this setting is used in the following experiments.

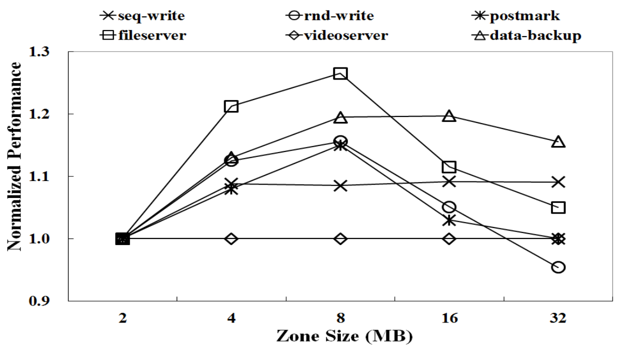


Fig. 4. Normalized performance under various zone sizes (zone scheme)

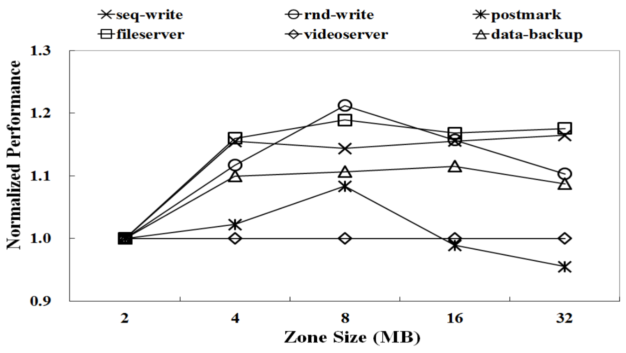


Fig. 5. Normalized performance under various zone sizes (zone_segment scheme)

As mentioned in Section III-A, in the *zone_segment* scheme, the threshold α has an impact on whether the zones with the maximum *ASL* value should be selected as the target zones. If the value of α is extremely large, the *zone_segment* scheme would retrograde to the *zone* scheme. On the contrary, an extremely small α value cannot efficiently reduce seeks between short segments, leading to poor writeback performance. Fig. 6 shows the performance results under different values of α. The results are normalized to the performance of the *zone_segment* scheme that does not consider the α value (i.e., setting α as 0). According to Fig. 6, setting α as 50 achieves good performance under all the workloads, and hence the value is used in the other experiments.

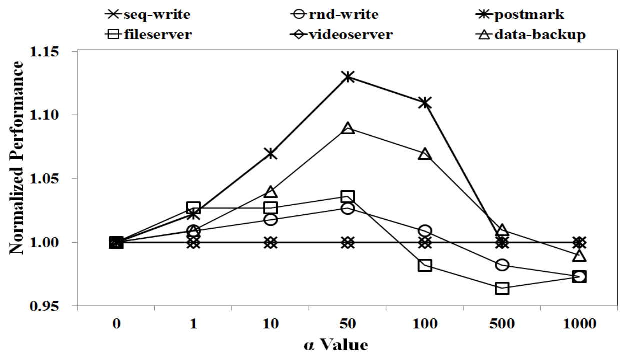


Fig. 6. Normalized performance under various α values

D. Performance of Different Schemes in TESA

Fig. 7 compares the performance of the two schemes used in the TESA writeback policy. The results are normalized to those under the *zone* scheme. As seen in Fig. 7, the *zone_segment* scheme outperforms the *zone* scheme by up to 15.4%. As mentioned in Section III-A, the *zone_segment* scheme tries to flush a zone containing long segments and thus it reduces not only the seek distance but also the number of seeks, leading to a better performance compared to the *zone* scheme. According to the results, the *zone_segment* scheme is used when comparing TESA with the other writeback policies.

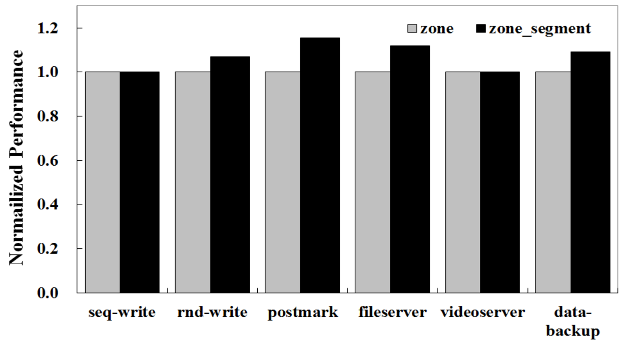


Fig. 7. Performance comparison between the zone and zone_segment schemes

E. Performance of Different Writeback Policies

Fig. 8 compares the performance of different writeback policies: the file based writeback policy used in Linux, the recency based policy that considers only the temporal information of the dirty buffers, a DULO-like policy that flushes the largest segment among the non-recently-used buffers [7], and the TESA policy. These policies are referred to as the *file*, *recency*, *segment*, and TESA policies, respectively. The results are normalized to those of the *file* policy. As shown in Fig. 8, TESA outperforms the *file* policy by up to 33.1%. This is because TESA reduces disk positioning time and additional write traffic by considering both temporal and spatial information. When compared with the *recency* policy, which considers only temporal information, TESA has a performance improvement of up to 21.0%. Compared to the *segment* policy, which also considers both temporal and spatial information, TESA has a performance improvement of up to 10.9% since it tries to flush nearby segments (i.e., segments in a zone). Under the sequential write workload (i.e., *seq-write*), all the policies have similar performance because all of them lead to sequential data flushes. Under the read intensive workload (i.e., *videoserver*), these policies also result in similar performance.

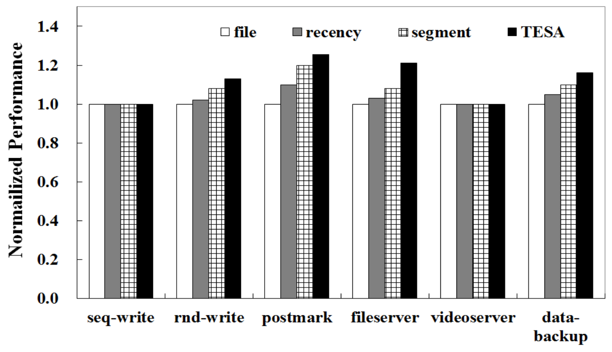


Fig. 8. Normalized performance of different writeback policies

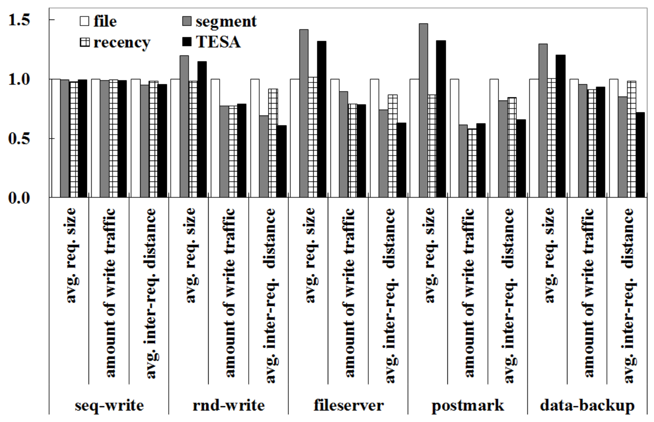


Fig. 9. Normalized performance factors of different writeback policies

To further understand the performance of the writeback policies, the disk driver was modified to measure the average request sizes, the total numbers of blocks written to the storage, and the average inter-request distances, during the execution of each of the write dominated workloads (i.e., *seq-write*, *rnd-write*, *postmark*, *fileserver* and *data-backup*) under the writeback policies. Note that the inter-request distance is defined as the distance between the last block of a request *R* and the first block of the request following *R*. Fig. 9 shows the results normalized to those under the *file* policy. According to Fig. 9, the TESA policy results in increased average request size when compared to the *file* and *recency* policies since TESA tends to flush zones with large segments. Flushing larger segments often leads to superior performance for disk. Compared to the *segment* policy which greedily flushes the largest segment, TESA shows only a small reduction in the average request size (i.e., 5.7% reduction in average). Moreover, the *recency*, *segment* and TESA polices all reduce the total write traffic to the storage when compared to the *file* policy. This is due to that these policies consider the temporal information of the buffers and can reduce the frequency of flushing recently-updated buffers. Finally, both *segment* and TESA policies result in reduced average inter-request seek distances than the other policies. This is because the both policies tend to flush large segments, and each segment can be divided into multiple requests with adjacent blocks. TESA achieves even shorter average inter-request distances than the *segment* policy (up to 19.5% reduction), which is contributed by flushing nearby segments (i.e., segments in a zone).

F. Memory Overhead of TESA

As mentioned before, TESA maintains in-memory data structures such as *zone_cb* and *seg_cb*, and it adds a field in the *buffer_head* structure. In the current implementation, the sizes of the *zone_cb* and the *seg_cb* structures are 1292 and 16 bytes, respectively. The added field in the *buffer_head* structure occupies 4 bytes. Table I shows the maximum memory overhead of TESA during the execution of the workloads. As shown in the table, the maximum memory overhead is less than 446.8KB for all the workloads. Such overhead is insignificant and would not lead to noticeable performance degradation.

TABLE I
MAXIMUM MEMORY OVERHEAD OF TESA

Workloads	Maximum Memory Overhead (KB)
seq-write	107.8
rnd-write	382.3
postmark	446.8
fileserver	241.1
videoserver	10.6
data-backup	299.5

V. CONCLUSION

In this paper, an intelligent writeback policy called TESA is proposed to improve the performance of flushing dirty buffers in home NAS devices. By taking both temporal and spatial information of the dirty buffers into consideration, the TESA policy reduces both the disk positioning time and the amount of disk writes.

The TESA writeback policy was implemented on a NAS evaluation board running Linux kernel. The performance of TESA was compared with several existing writeback policies under 6 benchmarks. According to the performance results, the performance of TESA is superior to the original file based writeback policy used in Linux by up to 33.1%. When compared to a recency based policy, TESA shows a performance improvement by up to 21.0%. Moreover, TESA outperforms a previous policy that also exploits both the temporal and spatial information of the dirty buffers by up to 10.9%.

REFERENCES

[1] J. T. Kim, Y. J. Oh, H. K. Lee, E. H. Paik, and K. R. Park, "Implementation of the DLNA proxy system for sharing home media contents," *IEEE Trans. Consumer Electron.,* vol. 53, no. 1, pp. 139-144, Feb. 2007.

[2] H. Sohn, Y. M. Ro, and K. N. Plataniotis, "Content sharing between home networks by using personal information and associated fuzzy vault scheme," *IEEE Trans. Consumer Electron.,* vol. 55, no. 2, pp. 431-437, May 2009.

[3] D. Díaz-Sánchez, F. Almenarez, A. Marín, D. Proserpio and P. A. Cabarcos, "Media cloud: an open cloud computing middleware for content management," *IEEE Trans. Consumer Electron.,* vol. 57, no. 2, pp. 970-978, May 2011.

[4] P. Bellavista, P. Gallo, C. Giannelli, G. Toniolo, and A. Zoccola, "Discovering and accessing peer-to-peer services in UPnP-based federated domotic islands," *IEEE Trans. Consumer Electron.,* vol. 58, no. 3, pp. 810-818, Aug. 2012.

[5] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proc. IEEE Int. Conf. Data Engineering (ICDE)*, pp. 3-12, Mar. 2000.

[6] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, "AWOL: an adaptive write optimizations layer," in *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 67-80, Feb. 2008.

[7] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 101-114, Dec. 2005.

[8] B. S. Gill and D. S. Modha, "WOW: wise ordering for writes - combining spatial and temporal locality in non-volatile caches," in *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 129-142, Dec. 2005.

[9] Y. Wang, J. Shu, G. Zhang, W. Xue, and W. Zheng, "SOPA: selecting the optimal caching policy adaptively," *ACM Trans. Storage*, vol. 6, no. 2, pp. 72-89, Jul. 2010.

[10] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *Proc. ACM Conf. Measurement and Modeling of Comp. Sys. (SIGMETRICS)*, pp. 241-251, May 1994.

[11] D. Anderson, J. Dykes, and E. Riedel, "More than an interface: SCSI vs. ATA," in *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 245-257, Apr. 2003.

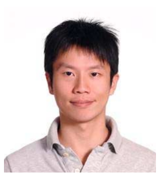
[12] J. Gim and Y. Won, "Extract and infer quickly: obtaining sector geometry of modern hard disk drives," *ACM Trans. Storage*, vol. 6, no. 2, pp. 46-71, Jul. 2010.

[13] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger, "On multidimensional data and modern disks," in *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pp. 225-238, Dec. 2005.

[14] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. Storage,* vol. 4, no. 2, pp. 114-169, May 2008.

[15] T. C. Huang and D. W. Chang, "VM aware journaling: improving journaling file system performance in virtualization environments," *Softw. Pract. & Exper.*, vol. 42, no. 3, pp. 303-330, Mar. 2012.

BIOGRAPHIES



Ting-Chang Huang received his BS, MS degrees in Computer Science at National Chiao Tung University, Hsinchu, Taiwan, ROC, in 2003 and 2005, respectively. He is currently a Ph.D. candidate in Computer Science at National Chiao Tung University, Hsinchu, Taiwan, ROC. His research interests include embedded systems, virtual machines, file and storage systems and power management techniques.

Da-Wei Chang (M'08) received his BS, MS, and PhD degrees in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan, in 1995, 1997, and 2001, respectively. He has been a postdoctoral researcher in National Chiao Tung University in 2002-2005, and an assistant professor in Electrical Engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan, in 2006. He is currently an associate professor in Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan. His research interests include operating systems, file and storage systems, virtual machines and embedded systems. He is a member of the IEEE and the ACM.