A Driver-Layer Caching Policy for Removable Storage Devices

Abstract

The popularity of flash memory will soon bring much attention to the criticism of file-system performance over flash memory. This work is motivated by the modularity designs in operating system components, such as bus and device drivers. We propose a filter-driver-layered caching design to resolve the performance gap among file systems and to improve their performance with the considerations of flash memory characteristics. An efficient hybrid tree structure is presented to organize and manipulate the intervals of cached writes. Algorithms are proposed in the merging, padding, and removing of the data of writes. The effectiveness and data consistency of the proposed approach are shown with some analysis study of FAT-formatted and NTFS-formatted USB flash disks. The proposed cohesive caching policy was implemented as a filter driver in Windows XP/Vista for performance evaluation. In the experiments, more than 10 times of performance improvement was achieved in many cases, when the cache size was only 64KB.

1. Introduction

There are two major designs for flash memory: NAND and NOR. NAND flash memory is mainly for the implementation of storage systems, and NOR has good performance in reads and supports XIP (eXecute-In-Place) to run programs directly. As MLC NAND flash memory gains its momentum because of the cost issue, how to retain the system performance has become a challenging issue, especially when the capacity of flash-memory storage device grows very rapidly[1]. For example, the time to program one page of $MLC_{\times 2}$ flash memory is $800\mu s$, while that of SLC flash memory is only $200\mu s$ [4, 5]. The performance problem is further exaggerated due to the behavior of file systems in the maintenance of file meta data and directory information. A number of writes of small sizes quickly result in performance deterioration of flash-memory-based file systems. Because MLC flash memory can only accommodate a very limited number of erases over each block, all of the mentioned problems also have serious impacts on the endurance of flash-memory storage systems. Such observations motivate this research. That is to resolve the performance gap among file systems and to improve their performance with the considerations of flash memory characteristics.

A NAND flash memory chip consists of many blocks, and each block consists of a fixed number of pages. A block is the smallest unit for erases, while a page is the smallest unit for reads and writes. Each page of small-block (/large-block) SLC flash memory can store 512B (/2KB) data, and there are 32 (/64) pages per block. The configuration of $MLC_{\times 2}$ flash memory is the same as large-block SLC flash memory, except that each block is composed of 128 pages [5]. Because each page is write-once, we do not overwrite data on each update. Instead, data are written to free pages, and the old versions of data are invalidated (or considered as dead). The update strategy is called "out-place update". In other words, any existing data on flash memory could not be over-written (updated) unless its corresponding block is erased. The pages that store live data and dead data are called "valid pages" and "invalid pages", respectively. In the literature, there were a lot of excellent researches and implementations to explore different system architectures and layer designs, e.g., [10, 22, 27, 36, 39], some exploited large-scaled and energy-aware storage systems, e.g., [11, 15, 16, 41, 45], and some exploited data compression and endurance enhancement for flash-memory storage systems, e.g., [9, 12, 43]. Some exploited new indexing structures for databases over flash-memory systems, e.g., [23, 24, 37, 38, 44]. Researchers also considered how to improve the performance of NAND flash memory with a SRAM cache, e.g., [21, 26, 32, 34], where OneNAND by Samsung presented a simple but effective hardware architecture to replace NOR with NAND and a SRAM cache, e.g., [21, 34, 35].

In recently years, some excellent work [20, 25, 42] proposed to add a write buffer in flash-memory devices to improve their random write performance. However, the write buffer will increase the hardware cost of flash-memory devices (due to cost of RAM buffer), may damage the integrity of file systems (due to power failures), and is not aware of the characteristics of file systems. One recent research direction is to adopt NAND flash memory as the cache of hard disks or as the fast booting devices of operating systems, e.g., [1, 2, 3, 6, 18, 30, 40]. Terrell et al. [19] proposed a design of flash-memory storage systems with SRAM as its cache. Harari et al. [17] proposed to cache data of writes to flash-memory storage devices. In the approaches, cache is in the stor-

[1] There are two popular NAND flash memory designs: SLC (Single Level Cell) flash memory and MLC (Multi-Level Cell) flash memory. Each cell of SLC flash memory contains one-bit information, while each cell of $MLC_{\times n}$ flash memory contains n-bit information. The endurance of a block of $MLC_{\times 2}$ flash memory is only 10,000 erase counts, compared to the 100,000 erase counts of its counterpart of SLC flash memory.

age devices for device performance improvement. Besides, Bennett et al. [8] considered the merging of operations of logical block area (LBA) ranges to improve the system performance. Rosich et al. [33] proposed a control system to cache data of writes, where the control system merges adjacent data blocks. Few results considered file system behaviors or designs. Different from popular caching ideas proposed in the previous work [8, 17, 19, 33], we are interested in the reducing of the performance gap among different file systems and their performance improvement with the considerations of flash memory characteristics. This work is also motivated by the modularity designs in operating system components, such as bus and device drivers.

In this paper, we propose a driver-layer caching policy, referred to as the *cohesive caching*, between the device driver and bus driver at the host computer. The designs of bus and device drivers are considered as black boxes. An efficient hybrid tree structure is presented to organize and manipulate the intervals of cached writes. Algorithms are proposed in the merging, padding, and removing of the data of writes so that the amount of data written to the storage devices is much reduced. The time complexity in the searching, inserting, and deleting of any cached write in the cache is $O(\lg n)$, where n is the number of cached writes in the cache. With analysis of USB-based FAT file systems, we show the effectiveness of the proposed approach. The caching policy is implemented as a filter driver in Windows XP/Vista. In the experiments, more than 93% of the file copying time was eliminated for USB-based FAT flash-memory file systems in the copying of Linux image files, when the cache size was only 64KB.

The rest of this paper is organized as follows: Section 2 presents the system architecture and the motivation of this paper. Section 3 presents the design of the driver-layer cohesive caching policy and its manipulation algorithms. Section 4 presents the behavior analysis of the FAT file system and the USB mass storage device driver. Section 5 summarizes the performance evaluation. Section 6 is the conclusion.

2. System Architecture and Motivation

A storage medium is usually accessed by a host through a device controller, where a device controller supports primitive functions in accessing the corresponding medium, such as reads and writes. Products may or may not choose to pack storage media and their controllers together as devices (Please see Figure 1). Example products are $MemoryStick^{TM}$, $SecureDigital^{TM}$, $SmartMedia^{TM}$, and xD^{TM} (and their card readers), as shown in Figure 1(b). $CompactFlash^{TM}$, and USB Flash Drives(UFDs) are other example products with controllers inside (Please see Figure 1(a)). At the host side, a bus driver is to control a bus, e.g., the Universal Serial Bus (USB), to communicate with a device controller, and a device driver is an implementation of the control mechanism for the corresponding storage medium. Filter drivers are introduced by modern operating systems [31], such as Windows XP and Vista, to provide system designers more flexibility in providing additional functionality, beside existing designs offered by operating-system or hardware vendors.

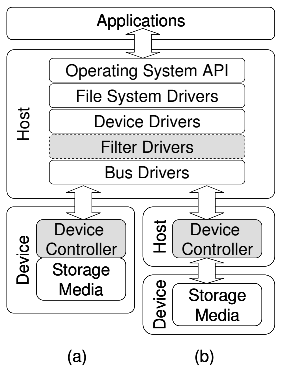


Figure 1. System Architecture

File System	NTFS	FAT32
Number of Written Files	19,535	19,535
Number of Written Directories	1,200	1,200
Number of write requests	24,513	179,670
Time taken (min:sec)	4:33	54:21
Number of Read Files	19,535	19,535
Number of Read Directories	1,200	1,200
Number of Reads	14,568	23,528
Time taken (min:sec)	2:53	3:19

Table 1. Write/Read files to/from a removable storage device over Windows XP

This work is motivated by significant performance differences in doing file manipulations over different file systems when the underlying storage medium is NAND flash memory (referred to as NAND as well). We are interested in NAND because it is a good example medium for removable storage devices, which must be of low cost and preferably huge in the capacity. It is also because NAND is a popular alternative for the storage system designs of embedded systems, due to its characteristics in non-volatility, low power consumption, low cost, and shock resistance. Table 1(Rows 2-5) shows our experimental results in writing 19,535 files in 1,200 directories to a 2GB Sandisk USB flash drive (UFD). The total size in the writing is 210MB. The average number of writes per file or directory is 1.18 in our experiments, when NTFS is adopted as its file system. The average number becomes 8.665 when FAT32 is adopted. The observation shows that the potential overheads introduced by writes over different file system can be very different. On the other hand, the average numbers of reads per file for the access of the same set of files remain quite different for NTFS and FAT32, as shown in Table 1(Rows 6-9). When close investigation was done, it was observed that NAND easily suffers from writes of small sizes. Such a phenomenon will exaggerate the performance problem in file-system manipulations , due to the handling of file meta information by the file system and the low transmission rate of a bus.

The technical question is how to resolve the performance gap among file systems and the management problems of NAND in a transparent way, without any modifications to bus and device drivers. Specifically, the problem is how much performance improvement is possible to FAT32 and even NTFS without considering the designs of bus and device drivers. In this research, the behavior of popular file systems, i.e., NTFS and FAT32, are investigated to drive the design of a good filter driver in system performance boosting. We consider file-system, device, and bus drivers as block boxes because of modular designs (and the drivers are usually provided in binary forms so that their modifications are often infeasible). Furthermore, a filter driver is a special layer that knows the characteristics of its corresponding storage medium (unlike file-system drivers) and runs at the host with better computing and memory resources (unlike the device controller). Such observations underlines the objective of this research. That is the design and implementation of a filter driver with a highly efficient caching policy. With considering the possibility to be adopted in embedded systems, the design of the caching policy should consider the system with restricted main memory and limited computing power.

3. A Driver-Layer Caching Policy - Cohesive Caching

3.1 Overview

In this section, a filter driver with a caching policy, referred to as *Cohesive Caching*, is proposed to improve the system performance of file systems over NAND. It is to be inserted between device drivers and bus drivers without any modification to existing driver designs, as shown in Figure 1. Strategies in request reprocessing, data caching, and housekeeping information maintenance will be proposed, where the concerns of limited memory usages of embedded systems should be considered. In this paper, we will focus our investigation over USB because of its extreme popularity as a bus interface for peripheral devices.

This filter driver consists of five components, as shown in Figure 2: *Dispatch Unit*, *Cohesive Cache*, *Transport Unit*, *Filesystem Identifier*, and *Debug Unit*. The driver intercepts, monitors, and creates I/O requests to any device under its control, e.g., those belonging to the USB mass storage device class (including USB flash drives and hard disks with their ATA/SATA to the

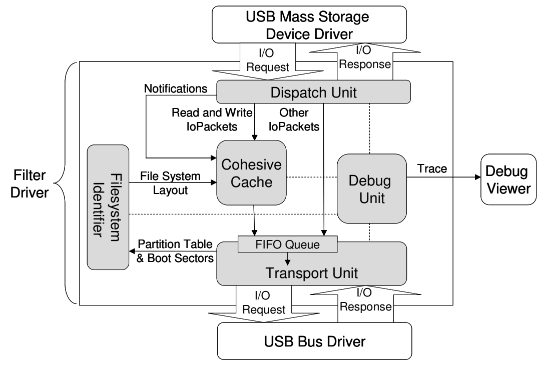


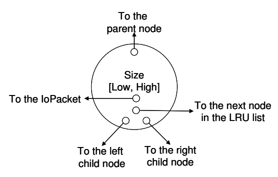
Figure 2. The system modules of the filter driver design

USB converter). The Dispatch Unit intercepts USB I/O requests from the USB device driver, converts them to IoPackets, and passes them to the Cohesive Cache or Transport Unit, according the types of the IoPackets. A revised data structure based on the "interval tree [13]" is proposed to cache IoPackets (Please see Section 3.2)) and to reprocess requests (Please see Section 3.3). It is not only to cache, merge, and retrieve cached data efficiently but also to reduce the number of IoPackets sent to removable storage devices. Data in the Cohesive Cache are flushed to the the Transport Unit in a Least-Recently-Used (LRU) fashion based on their referenced time. The Transport Unit converts IoPackets into USB I/O requests and sends them to the USB Bus Driver in a first-in-first-out fashion. The Filesystem Identifier retrieves the partition table from a removable storage device so as to determine the layout of file systems in the device, and then passes the information to the Cohesive Cache as a reference in request reprocessing (so as to reduce the number of IoPackets in the cache). The Debug Unit collects access patterns and runtime messages. It is to report the runtime information to a debugging viewer for debugging purposes.

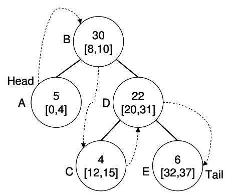
3.2 An LRU-Interval Tree

USB I/O requests are transformed into IoPackets by the Dispatch Unit and kept in the Cohesive Cache. We propose an LRU-enhanced data structure, referred to as an *LRU-interval tree*, based on the interval tree concept [13] to do request manipulations, such as merging, where an interval search tree is a balanced tree in which each node is associated with an interval for key searching.

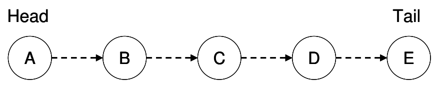
Each node in an LRU-interval tree denotes an IoPacket of data in continuous sectors, as specified by its corresponding Logical Block Address (LBA) interval $[Low, High]$, as shown in Figure 3(a). Each node is also associated with three links: The *left* and *right links* of the node point to subtrees with intervals to the left and right of its corresponding interval (i.e., intervals with LBA's less or larger than the LBA's of the corresponding interval),



(a) The node structure



(b) An example LRU-interval tree



(c) The corresponding LRU list

Figure 3. An LRU-interval tree

respectively. There should not be any overlapping among the intervals of nodes. The *LRU link* of each node, denoted as a dashed line, helps in the maintenance of a double link list based on the insertion time of each node into the LRU-interval tree, as shown in Figures 3(b) and 3(c). The attribute *size* of a leaf node is the number of sectors in the corresponding interval, i.e., the interval length $(High - Low + 1)$. The attribute *size* of an intermediate node is the sum of its interval length and those of all nodes in the subtrees pointed by its *left* and *right* links, as shown in Figure 3(b). The maintenance of the *size* values is to reduce the time to calculate the amount of cached data corresponding to each node and a subtree.

The time complexity in the insertion or deletion of a node in an LRU-interval tree is $O(\lg n)$, where n is the number of nodes in the tree. The searching of a node in an LRU-interval tree has a complexity $O(\lg n)$, where a traversal over comparisons of nodes'interval is done. When the cache is full, the removing of nodes based on their LRU order can be also done efficiently by traversing the LRU list and restructuring the resulted LRU-interval tree until the cache size is restored back to a safe level. Figure 3(b) shows an LRU-interval tree resulted from the insertions of five nodes $A[0,4]$, $B[8,10]$, $C[12,15]$, $D[20,31]$, and $E[32,37]$ in order. The corresponding LRU list (linked up by dashed lines) is as shown in Figure 3(c). The system requires that the packet size of each IoPacket should not be over the maximum acceptable packet size P_{max}, as defined by the USB mass storage protocol ($S_{max} = P_{max}/$ (the sector size)). As shown in the following section, no merging of nodes, i.e., IoPackets, should result in a node with a size larger than the maximum bound (Please see Section 3.3.1).

3.3 Cohesive Cache

The Cohesive Cache is mainly to cache data so as to reduce the amount of I/O requests sent to storage devices. It is to improve the file-system performance over tertiary storage devices, especially those over flash-memory storage devices. We propose to focus on the designs of data caching for writes, due to the fact that writes are much slower than reads over flash memory.

USB I/O requests are converted into IoPackets by the Dispatch Unit and sent to the Cohesive Cache. The *Caching Procedure* is invoked to insert IoPackets into the LRU-interval tree (Please see Section 3.3.1) for request manipulations. The *Trimming and Merging Procedure* and the *Padding and Merging Procedure* are then invoked to reduce the number of IoPackets sent to storage devices (Please see Section 3.3.2). Any interval overlapping of IoPackets in the LRU-Interval tree should be checked up for merging, padding, and removing. When the cache is full, IoPackets should be flushed to the proper storage device (through the Transport Unit) in an LRU fashion until the size of the cached data is no more than a pre-determined threshold.

Note that the USB device driver usually issues the "Allow Medium Removal" command to declare the end of a series of I/O requests. Once the Dispatch Unit receives the "Allow Medium Removal" command, it should notify the Cohesive Cache to flush out all of the cached data so that the data consistency is preserved. In case the "Allow Medium Removal" command is only issued after the unmounting of the disk in some systems, the Cohesive Caching should flush all of the cached data in a regular frequency, e.g., once per second, especially when the device driver stops sending any read or write command to the Dispatch Unit for some time.

3.3.1 The Caching Procedure

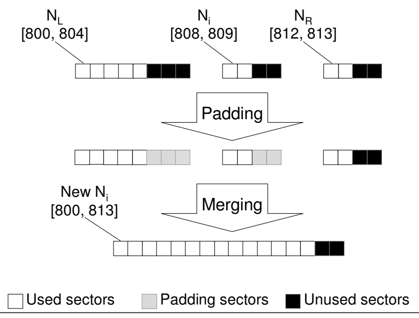


Figure 4. Writes over the cluster area.

The Caching Procedure is invoked whenever the Cohesive Cache is requested to insert the IoPacket P_i of a write request to the LRU-interval tree. Algorithm 1 shows the pseudo code of the Caching Procedure, where P_i, LIT, and CA denote the IoPacket (received from the Dispatch Unit), the LRU-interval tree, and the cluster areas of the file system, respectively. The objective is to do merging, padding, and removing over nodes (of IoPackets) of the LRU-interval tree to reduce writes to the storage devices. Note that many modern file systems have two storage partitions: The cluster and non-cluster areas. A write to a cluster in the cluster area might only occupy the first few sectors of the cluster such that the rest sectors are allocated but not used, where a cluster is of a fixed number of sectors, e.g., 8 sectors per cluster. In a non-cluster area, the smallest unit for space allocation and reclamation is one sector, instead of one cluster. Figure 4 shows three example write requests that correspond to nodes (i.e., IoPackets) N_L, N_i, and N_R with LBA intervals $I[N_L] = [800, 804]$, $I[N_i] = [808, 809]$, and $I[N_R] = [812, 813]$, respectively. Here $Low[N]$ and $High[N]$ in Algorithm 1 denote the lower and upper bounds of the LBA interval, denoted as $I[N]$, of a node N, respectively.

When an IoPacket, i.e., P_i, is received, the procedure first checks up whether caching is not enabled, or the size of P_i is larger than the maximal package size P_{max}. If so, the IoPacket is simply inserted into Q_{out}, i.e, the FIFO queue of IoPackets for the Transport Unit (Steps 1-2). If not, then Q_{out} is initialized and the corresponding node, i.e., N_i, is created (Steps 4-5). Note that Q_{out} is returned by the procedure and data in Q_{out} is eventually flushed to the storage device by the Transport Unit. Steps 6-8 repeatedly look for any node N_o that has some interval overlapping with N_i and invoke the *Trimming and Merging Procedure* (Please see Section 3.3.2) to merge them into N_i. Note that IoPackets that are created due to any constraints in merging adjacent nodes N_o and N_i are inserted into Q_{out}, i.e., the returned IoPackets from the *Trimming and Merging Procedure*, (Steps 6-8). An example constraint that prevents the procedure from node merging is the maximum data size of an IoPacket allowed by the USB mass storage protocol, i.e., 64KB or 128 LBA's (referred to as S_{max}). Whenever any IoPacket is created because adjacent nodes N_o and N_i cannot be merged into N_i, it should be flushed to the storage device (i.e., the inserting of the IoPacket returned by the *Trimming and Merging Procedure* into Q_{out}). It is required that no two nodes in the tree has any interval overleaping.

Depending on whether N_i is in a cluster area, the left and right adjacent nodes of N_i are looked up in the tree, denoted as N_L and N_R, respectively (Steps 9-15). Here $S_{cluster}$ denotes the number of sectors in a cluster, and the second and third parameters of the search procedure denote the lower and upper LBA bounds for overlapping intervals, respectively. The *Padding and Merging Procedure* is then invoked to merge N_L, N_R, and N_i together into N_i (Please see Section 3.3.2). Note that all of the adjacent nodes that cannot be merged into N_i will remain in LRU-interval tree (Steps 16-18). Consider nodes N_L, N_R, and N_i in Figure 4, N_L and N_i are padded with zeros and become $I[N_L] = [800, 807]$ and $I[N_i] = [808, 811]$ so that the three intervals can be merged as one longer interval $I[Ni] = [800, 813]$. N_R does not have its interval $I[N_R] = [808, 811]$ being padded because the padding operation is applied mainly to reduce the memory size of the tree. Steps 19-24 keep looping until the size of the cache is no more than the cache capacity bound, i.e., C_{max}. In each iteration, one LRU node is removed (Steps 20-21) and inserted into Q_{out} (Step 22) for flushing into the proper storage device.

Algorithm 1: The Caching Procedure

```
   Input: $P_i$, $LIT$, $CA$
   Output: $Q_{out}$
1  if $CacheEnable = False$ or $Size[P_i] > P_{max}$ then
2      $Q_{out} \leftarrow P_i$ ;
3  else
4      Reset $Q_{out}$;
5      Construct a node $N_i$ for IoPacket $P_i$;
       // Trimming and merging
6      while $N_o \leftarrow Search(LIT, Low[N_i], High[N_i]) \neq null$ do
7          $Q_{out} \leftarrow Q_{out} + TrimmingAndMerging(LIT, N_i, N_o)$;
8      end
       // Padding and Merging: Search adjacent nodes of $N_i$
9      if $I[N_i]$ is in CA then
           // Cluster area: Implicitly adjacent
10         $N_L \leftarrow Search(LIT, Low[N_i] - S_{cluster}, Low[N_i] - 1)$ ;
11         $N_R \leftarrow Search(LIT, Low[N_i] + 1, Low[N_i] + S_{cluster})$ ;
12     else
           // Non-cluster area: Adjacent
13         $N_L \leftarrow Search(LIT, Low[N_i] - 1, Low[N_i] - 1)$ ;
14         $N_R \leftarrow Search(LIT, High[N_i] + 1, High[N_i] + 1)$ ;
15     end
       // Padding and Merging: merge $N_L$ into $N_i$,then $N_R$ into $N_i$
16     $N_i \leftarrow PaddingAndMerging(LIT, A, N_i, N_L)$ ;
17     $N_i \leftarrow PaddingAndMerging(LIT, A, N_i, N_R)$ ;
       // Insert the node $N_i$ into LRU-interval tree
18     $InsertNode(LIT, N_i)$;
       // Maintaining the cache size
19     while $Size[N_{root}] > C_{max}$ do
20         $N_{vic} \leftarrow LRU(L)$ ;
21         $RemoveNode(LIT, N_{vic})$ ;
22         $Q_{out} \leftarrow IoPacket[N_{vic}]$ ;
23         Delete $N_{vic}$ without deleting its $IoPacket$;
24     end
25 end
26 return $Q_{out}$;
```

3.3.2 The Trimming, Padding, and Merging

The Trimming and Merging Procedure is to merge nodes N_o and N_i into N_i if their intervals overlap with each other. There are four overlapping conditions considered in the procedure, referred to as overlapping conditions 1-4 (Please see Figure 5). N_i is the node that is just inserted into the tree (Please see the previous section) so that the corresponding data of N_o is replaced with the data of N_i. If the first overlapping condition is satisfied, then the corresponding data of N_o is simply replaced with the data of N_i. If the second overlapping condition is satisfied, then the overlapped part of N_o is trimmed first to derive *RightChunk*. If the merging of *RightChunk* and N_i is no more than the maximum data size of an IoPacket S_{max}, then they are merged; otherwise, *RightChunk* is flushed to the proper storage device. The third overlapping condition is similar to the second condition (where the difference is only on the part being trimmed). If the fourth overlapping condition is satisfied, then the interval of N_i contains that of N_o such that N_o and its IoPacket are deleted.

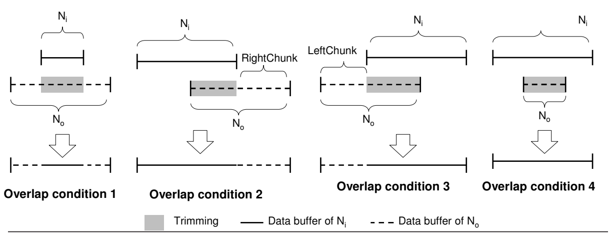


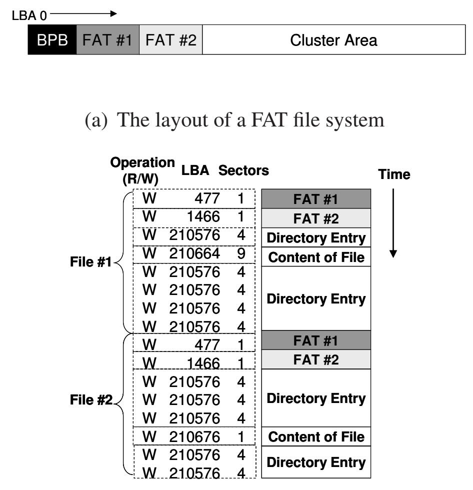
Figure 5. Overlapping conditions

The Padding and Merging Procedure is to merge two nodes together if the two nodes are adjacent to each other in the cluster or non-cluster area: N_i is the node that is just inserted into the tree (Please see Section 3.3.1), and N_n is a node that might be adjacent to N_i. As shown in Figure 4, N_n could be N_L or N_R. We should first identify the interval length for padding. If N_i and N_n are adjacent and the interval length of the merged node does not exceed the maximum data size of an IoPacket (i.e., S_{max}) after the padding operation, they are merged together. In general, we first merge N_L into N_i, and then the N_R is merged into the new N_i (Please see the example shown in Figure 4).

4. Behavior Analysis: FAT and the USB Mass Storage Device Driver

The purpose of this section is to explore the impacts of the proposed caching approach in practice. FAT and USB are selected in the analysis because FAT is the default file system for flash-memory-based storage devices, and USB is the most popular bus interface for peripheral devices. Note that NTFS is not recommended by Microsoft for removable storage devices [29], specifically flash-memory-based ones[14].

The layout of a FAT file system consists of four parts, as shown in 6(a): The boot parameter block (BPB), the primary file allocation table (FAT #1), the secondary file allocation table (FAT #2), and the cluster area. There are five major steps in the manipulation of a file: There are updates to two files illustrated in Figure 6(b), where "sectors" denotes the number of sectors being accessed by the corresponding operation, i.e., read or write (and the LBA of its first sector). They are updates to the primary FAT, the secondary FAT, the directory entry, the file contents, and the directory entry. Sectors are updated



(a) The layout of a FAT file system

(b) A access trace of a FAT file system

Figure 6. The behavior of a FAT file system

excessively to the directory entries even if the "Optimize for performance" option is enabled in the device manager of the operating system. One major reason for such an extreme approach comes from the fact that the file system issues a write request for updating a directory entry filed. It results in extremely lower system performance and serious endurance problems for NAND flash memory[2].

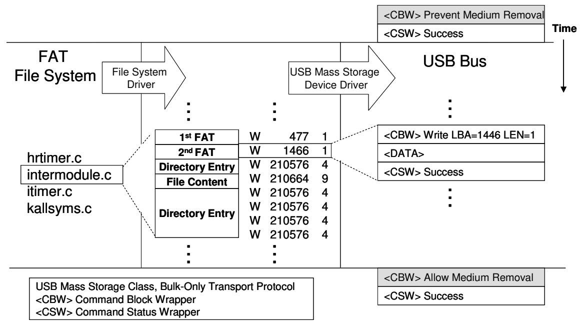
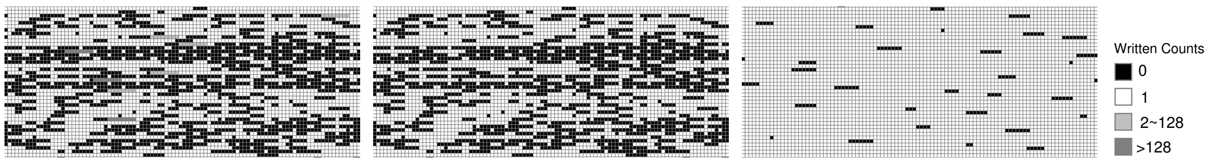


Figure 7. An example in writing files through the USB driver stack

The problems are further worsened by the considerations of the USB protocol implementations. As shown in Figure 7, four files are written to a USB-based NAND flash memory device. Each write request is done by a data request command (issued by the USB mass storage device driver). Each data request command consists of three phases: Command, data, and status phases. In each phase, a USB request block (URB) is created to carry the

[2] The time to program one page of Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND flash memory devices are about 200 μs and 800 μs, respectively. Each block of SLC and MLC$_{x2}$ NAND flash memory can only be erased for 100,000 and 10,000 times, respectively. [5, 7]



(a) No caching (the number of write requests: 179,670, 100%) (b) LRU caching (the number of write requests: 22,516, 12.53%) (c) Cohesive caching (the number of write requests: 4,879, 2.72%)

Figure 8. The access patterns of a storage device for a number of writes to a USB-based FAT file system. Note that each sub-figure shows sectors from LBA 95,300 to LBA 99,400, located in the cluster area (row-major, 100 LBAs/row, 1 LBA = 1 sector).

contents of the phase [28]. In the command phase, the URB carries a command block wrapper (CBW), which has the operation code of the request command, to the underlying USB bus driver. In the data phase, the URB receives/sends requested data from/to the USB bus driver where the request command is a read or write. In the status phase, the URB carries a command status wrapper (CSW) to receive the results of the request command from the underlying USB bus driver. Substantial overheads are thus resulted. Note that two control request commands “Prevent Medium Removal” and “Allow Medium Removal” are issued by the USB mass storage device driver before and after the series of writes, as shown in Figure 7, where the ”Allow Medium Removal” command is to declare the end of a series of I/O requests such that flushing is done for data consistency. Since FAT file systems do not know that the issuing of the command by the USB mass storage device driver, multiple duplicates of writes to the directory entries are issued by the file systems. The cohesive caching policy is implemented between the USB mass storage device driver and the USB bus driver. It can help in better integration among the file system and the drivers, and help the considerations of NAND characteristics. For example, the ”Allow Medium Removal” and ”Prevent Medium Removal” commands can be used to trigger caching and flushing of data issued by the file systems/drivers, respectively.

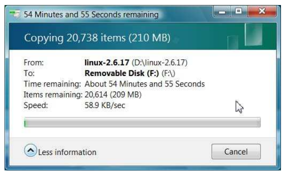
Consider the experiments done in Section 2 in the writing of 19,535 files to a removable storage device, as shown in Table 1. Figure 8 shows the access patterns of the experiments over a device with and without caching, where each cluster of the FAT file system consists of eight sectors. In the figure, black, white, and gray colors are for sectors that are never written, written only once, and written frequently, respectively. Figure 8(a) shows the access pattern when no caching is provided. There are 179,670 writes because sectors of directory entries are written very frequently. When an LRU caching is adopted, the number of writes is reduced substantially to 22,516, i.e., ***12.54%*** of that of no caching. The sectors of directory entries are no longer written so frequently. Those sectors are not gray-colored any more. When cohesive caching is adopted, the number of writes is again significantly reduced again to 4,879, i.e., ***2.72%*** of that of no caching because write requests with overlapped intervals are merged or removed! There are much less black-colored sectors because of padding in clusters. Note that padding in clusters slightly increases the amount of data actually transmitted to the removal storage devices; but it is considered very useful in performance improvement because access is done in clusters anyway.

5. Performance Evaluation

5.1 Performance Metrics and Experiment Setup

The purpose of this section is to evaluate the capability of the driver-layer cohesive caching policy over FAT file systems (Sections 5.2-5.5) and NTFS file systems (Section 5.6), in terms of performance improvement (Sections 5.2-5.4) and the ideal cache size (Section 5.5). The performance improvement was based on the number of write requests, the data transmission time, and the amount of data transmitted to a removable storage device. The ideal cache size for the cohesive caching policy was based on the number of write requests to removable storage devices.

The proposed cohesive caching policy was implemented as a filter driver and installed in Windows XP/Vista operating systems. The capability of the proposed policy was evaluated over some representative cases, realistic cases, and popular benchmarks. The representative cases were designed and generated to evaluate the performance of the proposed policy with different numbers of directories and files, when the total size of files in an archive was the same. The realistic cases were to evaluate the performance of the proposed cohesive caching policy with some real archives, such as linux-kernel source codes, photos, MP3s, and videos. Benchmarks, such as $FDBench^{TM}$ and $Sandra^{TM} RemovableStorageBenchmark$, were used to further evaluate the performance of the proposed policy based on the industry practice. Note that for the representative cases and realistic cases, the experiments were conducted via the shell API of Windows Vista (i.e., Windows shell) with the “optimize for performance" option enabled. Figure 9(a) and Figure 9(b) are the transmission dialogs (of the Windows shell) in the copying of an archive (i.e., the archive used in Table 1) to a FAT-formatted UFD without and with the the proposed caching policy, respectively.

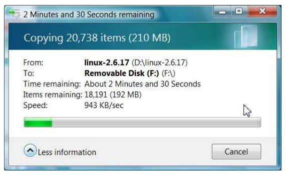


54 Minutes and 55 Seconds remaining

Copying 20,738 items (210 MB)

```
From:            linux-2.6.17 (D:\linux-2.6.17)
To:              Removable Disk (F:) (F:\)
Time remaining:  About 54 Minutes and 55 Seconds
Items remaining: 20,614 (209 MB)
Speed:           58.9 KB/sec
```

(a) No caching



2 Minutes and 30 Seconds remaining

Copying 20,738 items (210 MB)

```
From:            linux-2.6.17 (D:\linux-2.6.17)
To:              Removable Disk (F:) (F:\)
Time remaining:  About 2 Minutes and 30 Seconds
Items remaining: 18,191 (192 MB)
Speed:           943 KB/sec
```

(b) Cohesive caching

Figure 9. The transmission dialog in the copying of an archive to an FAT-formatted UFD

In the experiments, a 2GB USB flash drive (UFD) of SanDisk^{TM} Cruzer Crossfire^{TM} was under investigation. In order to prevent the uncertainty in allocating free spaces by file systems, we formatted the UFD before the transmission of each archive from the hard disk to the UFD. The cache size used for representative cases, realistic cases, and benchmarks was **64 KB**. The experiments evaluated three different cache settings: *no caching*, *caching*, and *cohesive caching*. No caching is the primitive policy of Windows Vista with the "optimize for performance" option enabled. Caching is an LRU caching mechanism with better performance in the manipulation of the LRU list because an balanced tree is implemented to index the intervals/packages cached in the LRU list. Cohesive caching is the proposed caching policy in this research.

5.2 Performance Improvement for the FAT File System: Case Studies

Three representative cases were designed for the experiments over an FAT-formatted UFD, and each case was classified as one class, i.e., **Class A**, **Class B**, and **Class C**. Each class consisted of ten archives, and each archive in the same class had the same size, excluding meta-data of archives. Meanwhile, each file in the same archive had the same size. The root directories of archives in Class A contained different numbers of files, ranging from 1,000 to 10,000 and stepped by 1,000, and the size of each archive was 100 MB. The root directories of archives in Class B had different numbers of directories, ranging from 1,000 to 10,000 and stepped by 1,000, but there was no file in the archives so that the size of each archive was 0 MB. Each archive of Class C had 1,000 second-layer directories in its root directory. Each second-layer directory contained one third-layer directory, and each third-layer directory stored one fourth-layer directory. This nested structure repeats until the number of directories, except the root directory, reaches the pre-defined number (that ranges from 1,000 to 10,000 and stepped by 1,000); each leaf directory, i.e., the deepest-layer directory, then stored with one file where the amount of size of the 1,000 files was 10 MB, i.e., the size of each archive was 10 MB.

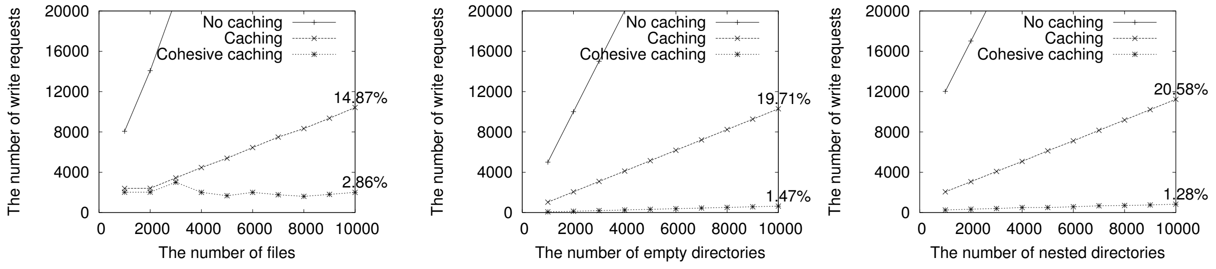
The intention of Class A was to evaluate the performance of caching policies for small files; Class B was to evaluate the performance of caching the meta-data of archives, i.e., directories and file information, excluding the contents of files; Class C was used to evaluate the performance of caching both small files and meta-data of archives.

5.2.1 The Number of Write Requests

Figure 10 shows that the proposed cohesive caching policy significantly reduces the number of write requests to the removable storage devices, where x-axis denotes the number of files or directories in an archive, and y-axis denotes the number of write requests. In Figure 10(a), the number of write requests of no caching grew extremely fast as the number of files in an archive grew and went beyond the scope of the figure. Compared to that without caching, the numbers of write requests for caching and cohesive caching were 14.87% and 2.86% of that of no caching, respectively, when the number of files was 10,000. Furthermore, caching and cohesive caching reduced more than 80% and 95% of write requests, respectively, when the number of files in an archive was larger than 5,000. However, the number of write requests for caching was still proportional to the number of files while that of cohesive caching remained bounded in a range. Comparatively, *the number of write requests for cohesive caching was less than $\frac{1}{5}$ of that of caching, when the number of files in an archive was 10,000.* We should point out that the cohesive caching had the worst performance when the number of files was 3,000. When the number of files in an archive was 3,000, the size of each file was around 33.3 KB. This prevented cohesive caching from merging the contents of any two files together because the maximal packet size of a USB command was 64 KB. The results of Figures 10(b) and 10(c) were similar to those of Figure 10(a) regardless of whether the archive contains "directories only" or "both directories and small files." *It is worth of noting that the number of write requests of cohesive caching was very close to $\frac{S_{ds}}{S_{max}}$ that was the minimum number of write requests through USB interface, where S_{max} denoted the maximum size of an write request accepted by the USB mass storage protocol, and S_{ds} denoted the space for both data and meta-data of an archive.*

5.2.2 The Data Transmission Time

Figure 11 shows the performance improvement of the proposed cohesive caching policy, in terms of data transmission time. Figure 11(a) shows the data transmission time of archives in Class A. The data transmis-



(a) Files in archives (Class A, Size: 100 MB)

(b) Directories in archives (Class B, Size: 0 MB)

(c) Files with nested directories in archives (Class C, Size: 10 MB)

Figure 10. Comparison on the number of write requests for different numbers of files or directories in an archive (FAT)

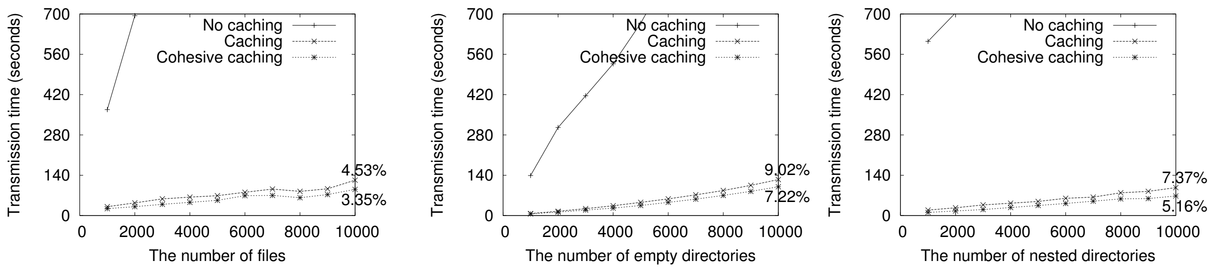
sion time increased as the number of files increased. When the number of files was 10,000, the data transmission time for no caching, caching, and cohesive caching were 2713, 123, and 91 seconds, respectively. In other words, the data transmission times of caching and cohesive caching were 4.53% and 3.35% of that of no caching, respectively. Therefore, the performance of cohesive caching was 29 times and 1.35 times of the performance of no caching and caching, respectively, when the number of files was 10,000. The trends of the data transmission times of archives in Classes B and C were similar to those of Class A. Figure 11(b) shows that the performance of cohesive caching was 13.85 times and 1.25 times of the performance of no caching and caching, respectively, when the number of directories was 10,000. In Figure 11(c), it shows that the performance of cohesive caching was 19.37 times and 1.43 times of the performance of no caching and caching, respectively, when the number of nested directories was 10,000.

5.2.3 The Amount of Transmitted Data

Figure 12 shows the amount of data that were actually transmitted to the UFD, i.e., a removable storage device. when the size of a sector was 512 bytes, and the size of a cluster was 4 KB. As shown in Figure 12(a), the amount of sectors actually transmitted to the UFD grew rapidly as the number of files increased, when no caching was adopted. In the case of transmitting 10,000 files of an archive whose size was 100 MB, 540,692 sectors (264MB) were transmitted to the UFD when no caching was adopted. In this case, the overhead was more than 150%. Comparatively, caching and cohesive caching introduced limited overhead on transmitting of archives of 100 MB when the number of files in an archive varied from 1,000 to 10,000 (stepped by 1,000). We should point out that cohesive caching transmitted more sectors than caching did, because cohesive caching padded zeros to sectors that were allocated but not used in a cluster in order to merge two adjacent intervals(/IoPackets) together (Please see Section 3.3.1). Therefore, the more sectors allocated but not used, the more extra sectors transmitted by cohesive caching. However, it is interesting to see that when the size of each file was a multiple of the size of a cluster, the number of sectors transmitted by cohesive caching was almost the same as the number of sectors transmitted by caching. For example, when the number of files in an archive was 5,000, the size of each file was 20 KB, which was 5 times of the size of a cluster (4 KB), so that the number of sectors transmitted by cohesive caching was almost the same as the number of sectors transmitted by caching, as shown in Figure 12(a).

Figure 12(b) shows the transmission results of Class B. The number of transmitted sectors were proportional to the number of directories in an archive, and the number of sectors transmitted by cohesive caching was almost the same as the number of sectors transmitted by caching. The reason was because space allocation for the storing of meta-data was in the unit of one sector, instead of one cluster. Therefore, both the numbers of sectors transmitted by cohesive caching and caching were around one-third of that of sectors transmitted by no caching. Figure 12(c) shows the transmission results of Class C, and it was similar to Figure 12(b), except that the number of sectors transmitted by cohesive caching was larger than the number of sectors transmitted by caching for 3% to 12%. That was because the space allocation to store contents of files was in the unit of one cluster and therefore some sectors that were allocated but not used were padded by cohesive caching.

We should point out that even though cohesive caching transmitted more sectors than caching did in most cases, cohesive caching still spent less transmission time than caching did. That was because cohesive caching merged adjacent data (/intervals) together to reduce the number of write requests to the UFD. The fewer the write requests were issued to the removable storage device through the USB interface, the less overhead was introduced by the USB mass storage protocol. Furthermore, the address translation mechanism implemented in the UFD was not at the page level. The more write requests



(a) Files in archives (Class A, size: 100 MB) (b) Directories in archives (Class B, size: 0 MB) (c) Files with nested directories in archives (Class C, size: 10 MB)

Figure 11. Comparisons on the data transmission time for different number of files or directories in an archive (FAT)

to the UFD, the more overheads would be introduced to the management of the flash memory in the UFD, e.g., live-page copyings and block-erasings.

5.3 Performance Improvement for FAT File Systems: Realistic Cases

In order to further verify the performance improvement of the proposed cohesive caching policy for an FAT-formatted UFD, four realistic cases were under investigation. Without loss of generality, each case was chosen to represent one popular file format, and the archive size of each case was around 205 MB. The four cases were source codes of linux kernel, photos, MP3 files, and videos, and their characteristics are shown in Table 2. As shown in Table 3, cohesive caching outperformed no caching and caching in all cases. For instance, the data transmission times of no caching, caching, and cohesive caching for the source code of linux-2.6.17 were 3,261, 284, and 213 seconds, respectively. In other words, the data transmission time of cohesive caching to transmit the linux-2.6.17 source code was 6.53% and 75% of those of no caching and caching, respectively. On the other hand, the number of write requests for cohesive caching to transmit the linux-2.6.17 source code was only 2.7% and 21.7% of those of no caching and caching, respectively. We should point out that the performance improvement of the proposed cohesive caching policy was more significant when an archive contained more files and directories. In addition, the performance of cohesive caching was close to the optimum because the number of write requests issued by cohesive caching was very close to the theoretical minimum number of write requests, i.e., $205 \times 1024KB/S_{max} = 3,280$ where $S_{max} = 64KB$. The experimental results for the data transmission time and the amount of data actually transmitted to the removable storage devices were not included because they were similar to those in representative cases (Sections 5.2.2 and 5.2.3).

Archive	Archive Size	Number of directories	Number of files	Average file size
linux-2.6.17 source	210 MB	1,201	19,536	11 KB
Photo	205 MB	1	214	985 KB
MP3	206 MB	5	47	4 MB
Video	208 MB	1	2	104 MB

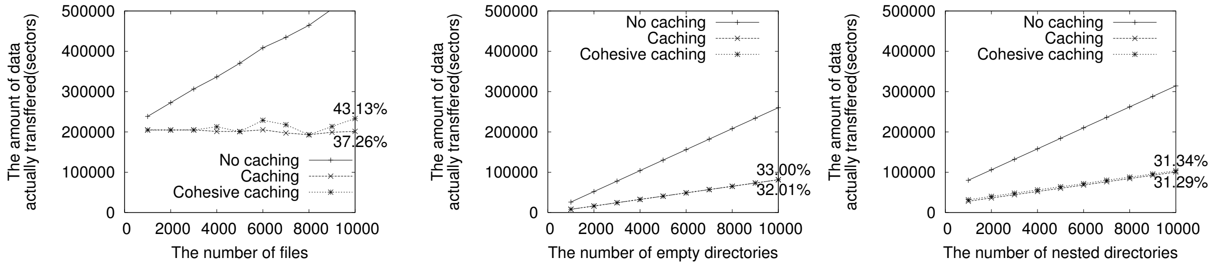
Table 2. The Archives of Some Realistic Cases

Archive	No Caching	Caching	Cohesive Caching
linux-2.6.17 source	179,670 (3,261 seconds)	22,488 (284 seconds)	4,879 (213 seconds)
Photo	5,022	3,906	3,431
MP3	3,841	3,630	3,506
Video	3,475	3,461	3,365

Table 3. Case Studies: Comparison on the number of write requests (FAT)

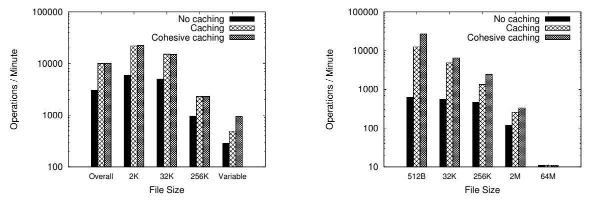
5.4 Benchmark Evaluation: FAT File Systems

Two popular benchmarks, i.e., *FDBench*^{TM} and *Sandra*^{TM} *Removable Storage Benchmark* (Referred to as Sandra for short), were used to certify the performance of the proposed caching policy for an FAT-formatted UFD, where FDBench was designed to evaluate the performance of flash-memory storage systems, and Sandra was designed to evaluate the performance of removable storage devices. Figure 13 shows the performance of caching policies reported from the benchmarks, where x-axis denotes the file sizes used for benchmarking, and y-axis denotes the number of operations per minute in the *logarithmic scale*. As shown in Figure 13(a), both cohesive caching and caching had much better performance than no caching did under FDBench. However, under FDBench, the performance of cohesive caching and caching was almost the same in most cases. That was because FDBench deleted copied files right after files were copied. This behavior introduced a lot of additional small writes for meta-data so that the contents (/data) of files could not be cached in the cohesive cache long enough to be merged effectively. Comparatively,



(a) Files in archives (Class A, size: 100 MB) (b) Directories in archives (Class B, Size: 0 MB) (c) Files with nested directories in archives (Class C, Size: 10 MB)

Figure 12. Comparison on the amount of data actually transmitted for different numbers of files or directories in an archive (FAT)



(a) FDBENCH (b) Sandra

Figure 13. Benchmark Evaluation (FAT)

Sandra didn't introduce too many additional small writes for meta-data since it only deleted written files at the end of the benchmarking. Therefore, the performance of cohesive caching on Standra was 11.95 times and 1.37 times of those of no caching and caching, respectively, when the size of tested files was 32 KB (as shown in Figure 13(b)).

5.5 The Ideal Cache Size for FAT File Systems

The proposed cohesive caching policy was implemented as a filter driver in Windows Vista, and the cache size of the filter driver could be configured through the system registry of Windows Vista. In order to evaluate the ideal cache size of the proposed cohesive caching for FAT-formatted UFD's, an archive was designed so as to introduce many write requests for meta-data and a lot of padding overheads for sectors that were allocated but not used. This archive included 20,000 files, where the size of each file was 5 KB. In other words, the size of the archive was 100 MB. In this archive, each file needed 2 clusters (8 KB) to store the file content (5 KB) so that it had 6 sectors (3 KB) being allocated but not used. Furthermore, since the size of each file in this archive was only 5 KB, the meta-data of these files must introduce a high percentage of overheads when this archive is copied to a removable storage device. As shown in Figure 14, the performance of cohesive caching became saturated when the cache size was no less than 64 KB. Comparatively, the performance of caching became saturated when the cache size was equal or larger than 8 KB. It was interesting to see that the performance of cohesive caching was worse than that of caching when the cache size was 8 KB or 16 KB. It was because the benefit from merging write requests with padding could not compensate the space overhead introduced by the padding. For instance, in order to cache the contents of two consecutive files in this archive, caching used 10 KB to cache them. Cohesive caching used 13 KB to cache them because of the padding of 6 sectors that were allocated but not used. Therefore, cohesive caching could not cache meta-data of files and merged write requests effectively, when the cache size was too small.

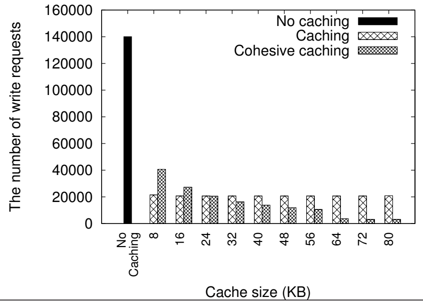
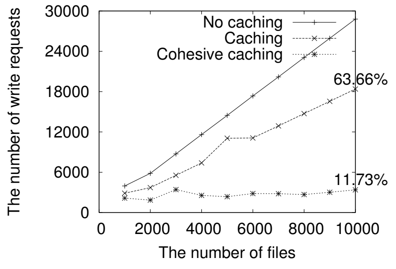


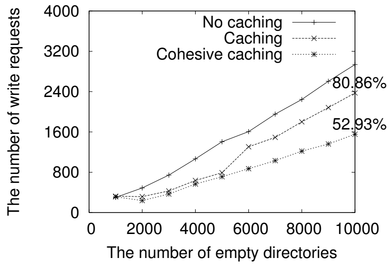
Figure 14. The Ideal Cache Size (FAT)

5.6 Performance Remarks for NTFS File Systems

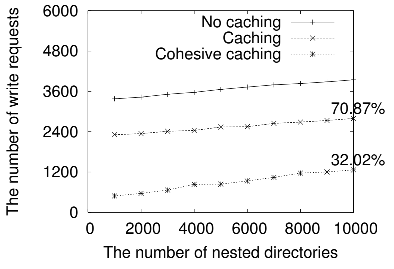
The experiment results over the NTFS-formatted UFD were similar to the trend of the results over the FAT-formatted UFD. The only difference is that the performance improvement of cohesive caching and caching was not as good as their corresponding ones over the FAT-formatted UFD, because meta-data management, space allocation, and index structure of NTFS file systems are better than those of FAT file systems. For example, for case studies, caching and cohesive caching only reduced the number of write requests (of no caching) for



(a) Files in archives (Class A, size: 100 MB)

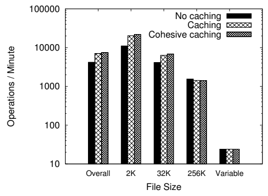


(b) Directories in archives (Class B, Size: 0 MB)

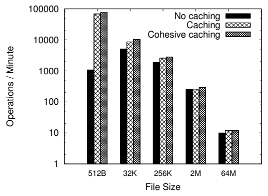


(c) Files with nested directories in archives (Class C, Size: 10 MB)

Figure 15. Comparison on the number of write requests for different numbers of files or directories in an archive (NTFS)



(a) FDBENCH



(b) Sandra

Figure 16. Benchmark Evaluation (NTFS)

36.33% and 88.37% respectively, when the number of files in an archive of Class A was 10,000 (as shown in Figure 15). In realistic cases, the transmission times of cohesive caching and caching for the linux-2.6.17 source were 85.71% and 93.4% of no caching respectively (as shown in Table 4). For benchmark evaluations, the performance of cohesive caching was over 1.07 times and 1.65 times (/1.1 times and 2.03 times) of caching and no caching in FDBench(/Sandra), respectively, when the tested file size was 32KB (as shown in Figure 16). As for the ideal cache size, with the same archive used in Section 5.5, the performance of cohesive caching became nearly saturated when the cache size was equal to 96 KB.

Archive	No Caching	Caching	Cohesive Caching
linux-2.6.17 source	60,457 (273 seconds)	41,105 (255 seconds)	8,268 (234 seconds)
Photo	3,951	3,972	3,776
MP3	3,489	3,480	3,452
Video	3,387	3,379	3,374

Table 4. Realistic Cases: Comparison on the number of write requests (NTFS)

6. Conclusion

This work is motivated by significant performance differences in doing file manipulations over different file systems over NAND flash memory. An efficient cohesive caching policy is proposed for removable storage devices. We propose a filter-driver-layered caching design to resolve the performance gap among file systems and to improve their performance with the considerations of flash memory characteristics and main-memory requirements. An efficient hybrid tree, called LRU-interval tree, is designed to organize and manipulate the intervals of cached write requests. With throughout analysis of the USB mass storage protocol, the time to flush cached writes to the underlying storage devices is determined to guarantee data consistency. In the experiments, more than 10 times of performance improvement was achieved in many cases, when the cache size was only 64KB.

For future research, we should further exploit the modularity design of filter-drivers to build up experimental platforms. The proposed filter-driver caching policy and design can also be further extended to integrated designs of devices with secondary/tertiary storage devices and flash memory.

References

[1] Flash Cache Memory Puts Robson in the Middle. *Intel.*

[2] Software Concerns of Implementing a Resident Flash Disk. *Intel Corporation.*

[3] Hybrid Hard Drives with Non-Volatile Flash and Longhorn. *Microsoft Corporation*, 2005.

[4] K9NBG08U5M 4G * 8 Bit NAND Flash Memory Data Sheet. *Samsung Electronics*, 2005.

[5] NAND08Gx3C2A 8Gbit Multi-level NAND Flash Memory. *STMicroelectronics*, 2005.

[6] Windows ReadyDrive and Hybrid Hard Disk Drives, http:// www.microsoft.com/whdc/device/storage/hybrid.mspx. Technical report, Microsoft, May 2006.

[7] K9NCG08U5M 64Gbit Flash Memory Datasheet. *Samsung Electronics*, 2007.

[8] A. D. Bennett, A. D. Bryce, S. Gorobets, and A. W. Sinclair. Non-volatile memory and method with block management system. *United States Patent No.7139864*, 11 2006.

[9] L.-P. Chang. On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems. *22nd ACM Symposium on Applied Computing (SAC'07)*, March 2007.

[10] L.-P. Chang and T.-W. Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.

[11] L.-P. Chang and T.-W. Kuo. An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems. In *ACM Symposium on Applied Computing (SAC)*, pages 862–868, Mar 2004.

[12] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design. *44th ACM/IEEE Design Automation Conference (DAC'07)*, June 2007.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[14] B. Dipert. Pick a card: card formats. *EDN Magazine*, 7 2004.

[15] Y. Du, M. Cai, and J. Dong. Adaptive Energy-aware Design of a Multi-bank Flash-memory Storage System. *Proceedings of the 11 IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, 2005.

[16] H. Gyu and N. Chang. Energy-aware Memory Allocation in Heterogeneous Non-Volatile Memory Systems. *International Symposium on Low Power Electronics and Design (ISLPED'03)*, 2003.

[17] E. Harari, R. D. Norman, and S. Mehrotra. Flash EEPROM system. *United States Patent No.6914846*, 5 2005.

[18] Y. Hu and Q. Yang. Dcd-disk caching disk: A new approach for boosting i/o performance. *ISCA,ACM*, 1996.

[19] I. James Richard Terrell and P. Beard. Flash memory system having memory cache. *United States Patent No.6026027*, 1995.

[20] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. In *IEEE Transactions on Consumer Electronics*, pages 485–493. IEEE, 2006.

[21] Y. Joo, Y. Choi, C. Park, S. W. Chung, E.-Y. Chung, and N. Chang. Demand Paging for OneNAND^{TM} Flash eXecute-In-Place. CODES+ISSS, October 2006.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *Proceedings of the 1995 USENIX Technical Conference*, pages 155–164, Jan 1995.

[23] T. Kgil and T. Mudge. FlashCache: a NAND flash memory file cache for low power web servers. *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, 2006.

[24] G. Kim, S. Baek, H. Lee, H. Lee, and M. Joe. LGeDBMS: a small DBMS for embedded system with flash memory. *Proceedings of the 32nd international conference on Very large data bases*, pages 1255–1258, 2006.

[25] H. Kim and S. Ahn. BPLRU : A buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 239–252, 2008.

[26] J.-H. Lee, G.-H. Park, and S.-D. Kim. A new NAND-type flash memory package with smart buffer system for spatial and temporal localities. *JOURNAL OF SYSTEMS ARCHITECTURE*, 51:111–123, 2004.

[27] J.-H. Lin, Y.-H. Chang, J.-W. Hsieh, T.-W. Kuo, and C.-C. Yang. A NOR Emulation Strategy over NAND Flash Memory. *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'07)*, August 2007.

[28] Microsoft. *Windows Driver Kit documentation*. Microsoft Coroperation, 10 2006.

[29] Microsoft TechNet. *How NTFS works*. Microsoft Coroperation, 3 2003.

[30] T. Nightingale, Y. Hu, and Q. Yang. The design and implementation of a dcd device driver for unix. *USENIX*, 6 1999.

[31] W. Oney. *Programming The Microsoft Windows Driver Model*, chapter 16, pages 773–800. Microsoft Press, 2 edition, 2003.

[32] C. Park, J.-U. Kang, S.-Y. Park, and J.-S. Kim. Energy-aware demand paging on nand flash-based embedded storages. ISLPED, August 2004.

[33] M. N. Rosich, E. S. Noya, and R. M. Arnott. System for controlling a write cache and merging adjacent data blocks for write operations. *United States Patent No.5551002*, 8 1996.

[34] Samsung Electronics. *OneNAND Features and Performance*, 11 2005.

[35] Samsung Electronics. *KFW8G16Q2M-DEBx 512M x 16bit OneNAND Flash Memory Data Sheet*, 09 2006.

[36] D. Woodhouse. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium*, 2001.

[37] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An Efficient B-Tree Layer for Flash-Memory Storage Systems. In *the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Feb 2003.

[38] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An Efficient R-Tree Implementation over Flash-Memory Storage Systems. In *ACM 11th International Symposium on Advances on Geographic Information Systems (ACM-GIS)*, Nov 2003.

[39] C.-H. Wu and T.-W. Kuo. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *IEEE/ACM 2006 International Conference on Computer-Aided Design (ICCAD)*, November 2006.

[40] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile Main Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.

[41] Q. Xin, E. L. Miller, T. Schwarz, D. D. Long, S. A. Brandt, and W. Litwin. Reliability Mechanisms for Very Large Storage Systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 146–156, Apr 2003.

[42] S. yeong Park, D. Jung, J. uk Kang, J. soo Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory.

In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, New York, NY, USA, 2006. ACM.

[43] K. S. Yim, H. Bahn, and K. Koh. A Flash Compression Layer for SmartMedia Card Systems. *IEEE Transactions on Consumer Electronics*, 50(1):192–197, Feburary 2004.

[44] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. *4th USENIX Conf. on Files and Storage Technologies (FAST 2005), San Francisco, CA, December*, 2005.

[45] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 177–190, 2005.