PAPER

PAW: A Pattern-Aware Write Policy for a Flash Non-volatile Cache

Young-Jin KIM[†a)], ***Member***, **Jihong KIM**[††], **Jeong-Bae LEE**[†††], ***and*** **Kee-Wook RIM**[†††], ***Nonmembers***

SUMMARY In disk-based storage systems, non-volatile write caches have been widely used to reduce write latency as well as to ensure data consistency at the level of a storage controller. Write cache policies should basically consider which data is important to cache and evict, and they should also take into account the real I/O features of a non-volatile device. However, existing work has mainly focused on improving basic cache operations, but has not considered the I/O cost of a non-volatile device properly. In this paper, we propose a pattern-aware write cache policy, PAW for a NAND flash memory in disk-based mobile storage systems. PAW is designed to face a mix of a number of sequential accesses and fewer non-sequential ones in mobile storage systems by redirecting the latter to a NAND flash memory and the former to a disk. In addition, PAW employs the synergistic effect of combining a pattern-aware write cache policy and an I/O clustering-based queuing method to strengthen the sequentiality with the aim of reducing the overall system I/O latency. For evaluations, we have built a practical hard disk simulator with a non-volatile cache of a NAND flash memory. Experimental results show that our policy significantly improves the overall I/O performance by reducing the overhead from a non-volatile cache considerably over a traditional one, achieving a high efficiency in energy consumption.

key words: *disk-based storage systems, non-volatile cache, NAND flash memory, pattern-aware write cache policy, I/O clustering, performance enhancement*

1. Introduction

For decades, hard disks have been the most popular mass storage in both server and mobile systems. Due to highly low cost per unit capacity, mobile systems such as PDA, PMP, and UMPC (ultra-mobile PC) also adopt hard disks as secondary storage. However, hard disks have poor I/O performance for random I/O requests because a seek time basically depends on the position variation of a disk head between successive I/O requests. Thus, there have been a lot of studies aiming at improvement of the I/O performance in disk-based storage systems. The representative is using a cache close to a disk in order to store the requested data with the expectation of their being accessed again in the near future. A modern disk controller tends to employ a volatile device like a DRAM as a read cache as well as a non-volatile one as a write cache [1].

Extensive studies on read caches are reported in the literature (especially at the page cache level), which seek to achieve significant I/O performance enhancement by filtering I/O requests directed to a disk more efficiently than a widely-used read cache algorithm, LRU [2]. However, management of write caches has been relatively less studied. Furthermore, the majority of disk I/O requests, which are filtered by a page cache in the operating system, are known as writes in the disk-based server systems [3]. Similar disk I/O request patterns are found to occur often in mobile computing environments [4]. Therefore, the research on a good write cache management policy is a crucial issue.

In this paper, we propose a simple but practical write cache policy for a NAND flash memory in disk-based mobile storage systems. The proposed policy redirects write requests to a non-volatile cache of a NAND flash memory or a disk according to access patterns with the aim of optimizing the overall I/O performance of a disk-based mobile storage system. The policy is designed to reduce the disk I/O latency by forwarding non-sequential write data towards a disk as well as to mitigate excessive write accesses to a NAND flash memory, thus resulting in fewer garbage collections. Furthermore, in order to strengthen the sequentiality of the I/O requests we polish up the policy in combination with an I/O clustering-based queue scheduling algorithm. We also investigate how to achieve high energy efficiency by our policy.

2. Non-volatile Write Cache

There are two main benefits of using a non-volatile cache (hereafter, we call this NVC) in disk-based storage systems. The first is to enable writing data to an NVC, thus obtaining fewer disk accesses. That is, an NVC can absorb not a few write accesses, serving as a write cache. Thus, an NVC can reduce the write latency of a disk. At the level of an operating system, updates to meta data in a file system occur frequently in small sizes, resulting in performance degradation due to large seek times from random data accesses on a disk. Using an NVC can be beneficial in mitigating such performance degradation in a disk-based system.

The second is to enable maintaining data consistency in the overall storage systems. Unlike a DRAM cache, even at

Manuscript received August 19, 2009.
Manuscript revised April 30, 2010.

[†]The author is with the Department of Computer Science and Engineering, SunMoon University, Kalsan 100, Tangjeong, Asan, 336–708, Korea.

[††]The author is with the School of Computer Science and Engineering, Seoul National University, 599 Gwanangno, Gwanak-gu, Seoul, 151–742, Korea.

[†††]The authors are with the Department of Computer Science and Engineering, SunMoon University, Kalsan 100, Tangjeong, Asan, 336–708, Korea.

a) E-mail: youngkim@sunmoon.ac.kr

DOI: 10.1587/transinf.E93.D.3017

power failure an NVC can retain data safely. For example, important data such as bank transaction data can be written to an NVC instead of a DRAM cache and be flushed to a disk later for data synchronization. As a result, using an NVC can be profitable in upraising reliability in a storage system.

3. Motivation

In last decades, a NAND flash memory has been a leading non-volatile device that competes with hard disks in mobile storage device markets. The popular usage of a NAND flash memory can be attributed to its gradually progressive technical innovations. A NAND flash memory has many advantages over hard disks such as fast I/O access time, lower-power consumption, and higher shock resistance [15] although it has still high cost per unit capacity. Thus, using a NAND flash memory as a write cache can be attractive because its non-volatility is useful in write caching and the cost is also acceptable if the NAND flash write cache has a limited capacity.

A NAND flash memory consists of many physical pages and several pages are clustered into blocks. In the NAND flash memory, reads and writes are processed by a unit of page and erases occur by a unit of block. Data is written to a NAND flash memory in a way of out-of-place update unlike a hard disk. That is, the original page is invalidated and the required data is written to a new page. This feature requires an address mapping software called a flash translation layer (FTL), which maps a logical block address from a host operating system into a pair of a physical page number and a physical block number [16], [17].

If a write access arrives and there is no free space (actually, the number of free blocks goes below a threshold value), blocks with some invalid pages should be recycled into free blocks. This process is called *garbage collection* (hereafter, we call this GC) and is also managed by an FTL. A GC policy selects a victim block for block reclamation. During a GC process, copying the valid data (pages) of a victim block to a free block and erasing the victim block are required. Hence, the overall GC overhead includes a read access time and a write access time multiplied by the number of valid pages, respectively, and an erase latency.

Since the GC overhead reaches several times of the write access time, frequent writes would be problematic for the overall I/O performance of a NAND flash memory. Especially when a NAND flash memory is used as an NVC, this phenomenon will become worse because all write operations go through the NVC and thus GC may occur repeatedly due to the limited size of the NVC, which is usually much smaller than that of a disk.

Therefore, in case of using a NAND flash memory as a write cache, it is highly important to consider the GC overhead, which may stem from the excessive writes due to the real I/O features of a NAND flash memory. However, existing works using a NAND flash memory as a write cache didn't investigate such overheads carefully. Our motivation comes from this point. In this paper, we suggest a pattern-aware write cache policy for a disk-based storage system with a NAND flash-based NVC, which is combined with an I/O clustering-based queuing method. Thus, we seek to find an efficient solution of mitigating the GC and write overheads in a NAND flash-based NVC, while suppressing the number of I/O requests directed to a disk in order to boost the overall I/O system performance in a disk-based storage system.

4. PAW: Pattern-Aware Write Cache Policy

4.1 Overview of PAW

In disk-based storage systems, there are many approaches to obtain performance enhancement at various levels of devices such as a cache, a queue, a controller, and media itself, and related software including a file system and a device driver. Figure 1 shows a logical view of a computing system with storage devices and illustrates that crucial performance-enhancing approaches at each level are highly related to cache management and queue scheduling [18].

Cache management is an effective performance-enhancing method by keeping data longer in a cache, which is likely to be used in the near future again regardless of the position of the cache. I/O requests, which reach a page cache at a file system level, will have different data attributes from those getting to a cache at a device level in the aspects of locality and I/O type. Thus, a well-devised cache management policy will have significant impact on the overall storage performance.

In the meantime, queue scheduling has the purpose of minimizing the average request response time by reordering I/O requests in the request queue at a device driver in the operating system or at a device controller within a device. Since the order of processed requests and the size of clustered requests have much influence on the performance of a storage system in terms of geometric operations, especially of a disk-based one, how well we schedule the requests in a queue is very important.

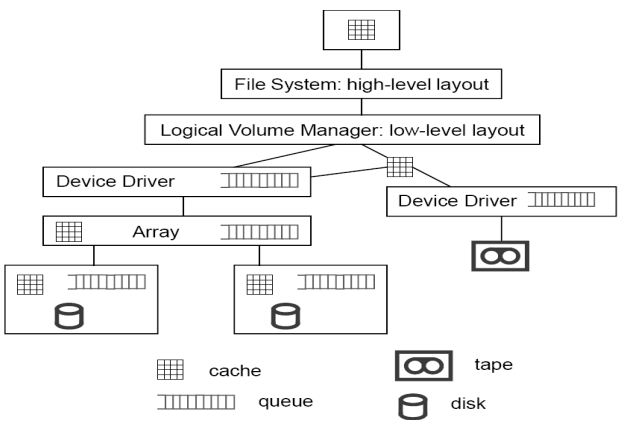

Fig. 1 Logical view of a system with storage devices [18].

Gill et al. proposed a performance-enhancing technique using wise ordering for writes called WOW. WOW is a write request ordering technique proposed for a non-volatile cache, combining a cache eviction policy and a queue scheduling method to achieve temporal locality and spatial locality together synergistically [1].

However, WOW doesn't consider the actual I/O latency of an NVC itself by assuming a volatile DDR memory to be an NVC. Since a DDR memory is quite different from a non-volatile device like a NAND flash memory in the aspect of I/O behaviors. For instance, a write access is processed within several nano-seconds for a DDR memory while it may be served for a few milli-seconds for a NAND flash memory (if there is no free block and thus garbage collection should occur, the worst latency may amount to about 1 second). Therefore, such assumption can cause an incorrect estimation of the processing time of I/O requests and impair the behavioral stability of the overall storage system by breaking synchronization between I/O operations in various devices. As a result, it will become difficult to evaluate and optimize exactly the I/O performance of a target storage system.

Our approach tries to remove the above irrationality by employing a NAND flash memory as an NVC and supplying a realistic I/O behavioral model for an NVC at the level of a storage device. It aims at obtaining the performance-enhancing potential from a synergistic combination of cache management and queue scheduling.

Recent studies show that typical mobile workloads have mixed access patterns of a large number of sequential accesses, big and small loop-type accesses, and a small amount of temporal accesses [26]–[28]. This is because applications in mobile devices have repetitive access patterns. For example, we execute an email program, a media player, a game program, a scheduling program, etc. circularly on a PDA. Such accesses will produce a mix of sequential accesses and random (or temporal) ones. Thus, it is important to deal with frequent sequential accesses together with random ones in mobile workloads since they might cause critical performance degradation of the overall system if they would not be faced.

To this end, we propose a practical pattern-aware write cache policy (PAW), which takes into account the access patterns of the load as well as the I/O features of a NAND flash memory in a disk-based mobile storage system to optimize the overall I/O performance. Then, we augment PAW with an I/O clustering-based queue scheduling algorithm in order to utilize better sequentiality of I/O requests and thus achieve a high efficiency in the performance as well as energy consumption.

Our contributions can be summarized three-fold. First, PAW is designed to face the characteristic of mobile workload patterns, which mainly consist of a mix of numerous sequential accesses and fewer non-sequential ones, in heterogeneous storage systems. Second, PAW employs a synergistic effect of combining a pattern-aware write cache policy and an I/O clustering-based queuing method to strengthen the sequentiality. Third, PAW tries to mitigate the I/O overhead of a NAND flash-based NVC, which mainly comes from GC operations occurring when many writes reach it, considering its real I/O features.

4.2 Algorithm of PAW

PAW is based on workload-aware load distribution, redirecting write requests to achieve high overall I/O performance in consideration of I/O access patterns. The rationales are that a disk and a NAND flash memory have different I/O behavioral features according to I/O access patterns and a large portion of I/O requests onto a disk is writes, as was mentioned previously. Figure 2 shows the overall structure of a proposed storage system with PAW, which consists of a device driver with a queue and a queue manager at a kernel level and a disk device with a disk controller, a DRAM, and a NAND flash-based NVC at a device level.

When a write I/O block access arrives, a disk controller checks if its access pattern is sequential or not and determines if it will write through the block to an NVC or a disk. Whether a required block is sequential or not depends on the continuity from the last LBA of a prior I/O request to the present one. If the LBA of an accessed block is continued to LBAs of pre-accessed blocks by a given threshold, the block is set as *sequential*. Otherwise, it is assigned *random*.

If a write access is random, it will be inserted into the MRU position of an NVC. Otherwise, the write will be forwarded to a disk. Therefore, redirecting data requests toward the NVC tends to absorb a considerable number of I/O accesses, which otherwise will be forwarded to the disk, and puts the disk to a lower-power mode, obtaining performance improvement and energy saving concurrently. When a dirty block is evicted from the NVC, it will be flushed to the disk. Otherwise, it will be discarded.

Our pattern-aware write cache management policy is polished with I/O clustering-based queuing in order to boost the overall performance as well as enhance the energy effi-

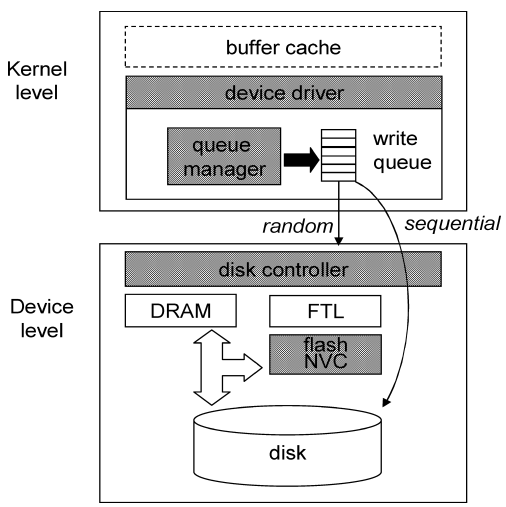

Fig. 2 Overall structure of a proposed storage system with PAW.

```
VARIABLES AND DEFINITIONS

PLUGGING_TIMEOUT: timeout value to unplug requests in the queue (i.e., request_queue)
MAX_REQ_QUEUE_SIZE: maximum size of request_queue
SEQUENTIALITY_THRESHOLD: threshold to determine if a request is sequential
request_queue: request queue
count: counter for checking insequential requests
lru_list: list for NVC cache management

QUEUING ALGORITHM

At the time when each request arrives,
  if (elapsed time of the beginning request of request_queue > PLUGGING_TIMEOUT)
    call service_ioreq
    //unplug the device with a current timestamp
  else
    Initialize count
    Traverse all requests to find sequentiality between the current request and a member request
      if a candidate is found, merge two requests
      else count++
    if (count == MAX_REQ_QUEUE_SIZE)
      call service_ioreq
      //unplug the device with a current timestamp
      insert the current request to the end of request_queue
    else
      insert the current request to the end of request_queue
```

```
PATTERN-AWARE WRITE CACHE MANAGEMENT

service_ioreq:
if (length of the request >= SEQUENTIALITY_THRESHOLD)
  Forward the request to a disk
else
  Forward the request to an NVC
  if (I/O type == READ)
    call read_NVC
  else
    call write_NVC

read_NVC:
if a cache hit occurs
  Read from pages until the requested I/O size is satisfied
  Update an lru_list
else // cache miss
  Read data from a disk
  Write data to a flash memory
    if valid pages are insufficient, invoke GC
  Update an lru_list

write_NVC:
if a cache hit occurs
  Write pages to a flash until the requested I/O size is satisfied
    if valid pages are insufficient, invoke GC
  Update an lru_list
else // cache miss
  Read data from a disk
  Write data to a flash memory
    if valid pages are insufficient, invoke GC
  Update an lru_list
```

Fig. 3 The PAW algorithm.

ciency of the storage system. Since I/O clustering strengthens the sequential property of each request in a request queue within a device driver [19], we conceive that the candidates of sequential ones among write accesses are sure to be more sequential and small writes, which might not be clustered together, are likely to have high temporal and low sequential localities.

With our scheme, write requests filtered at the request queue by the I/O clustering-based queuing method will arrive at a disk controller. The disk controller decides which write accesses are marked as sequential or random (i.e., non-sequential) according to their sequentialties. Random write requests will be forwarded to an NVC and sequential ones to a disk.

Figure 3 shows the detailed algorithm of PAW and related data structures. There are two parts in the PAW algorithm. First is the queuing algorithm and second is the pattern-aware write cache management routine. The queuing algorithm manages a request queue called *request_queue* to serve and schedule I/O requests in it. When each request arrives, the elapsed time of the beginning request member in this queue will be checked against *PLUGGING_TIMEOUT*, which variable is a time value to unplug (i.e., process) it. To count the number of merged requests, a variable *count* is used. If no requests are merged, that is, *count* reaches *MAX_REQ_QUEUE_SIZE*, the request with a current timestamp will be unplugged.

In the pattern-aware write cache management routine, unplugged requests are processed. If the sequentiality of a request reaches *SEQUENTIALITY_THRESHOLD*, it will be determined to be sequential and forwarded to a disk directly. Otherwise, the request will be forwarded to an NVC under the assumption of its having less sequentiality, that is, better randomness. It is natural that if requests have better randomness they will have more chances to have higher temporal locality. Thus, we expect that cache hits at the NVC will increase noticeably by storing request with less sequentiality to an NVC. To this end, PAW has two mechanisms to reinforce the sequentiality in dealing with I/O requests. First is to use an I/O clustering-based queuing method. Second is to use a threshold value in order to determine if a request is sequential or not.

5. Simulation and Results

5.1 Simulation Environment

We developed a trace-based I/O simulator which mimics the storage architecture shown in Fig. 2 and incorporated LRU and PAW. Our simulator models a device driver at a kernel level and an NVC and a disk at a device level. The device driver includes a queue manager which controls a request queue. As described previously, this queue is employed to realize I/O clustering by accumulating requests within it and merging some into a large one if sequentiality is found with the purpose of achieving the overall performance enhancement.

The hard disk model we used is MK4004GAH with a 1.8″ form factor and 4,200 RPM [25] and the NAND flash memory model used as an NVC is the K9K8G08U0M [21], which is a 1 G × 8 bit NAND flash with a 2 KB page size and a 128 KB block size. Table 1 shows the characteristics of our used hard disk and NAND flash memory. The disk's read (or write) latency is equal to the sum of the average rotation time and the average seek time. In this table, the

Table 1 Characteristics of the used hard disks and NAND flash memories [21], [25].

Device		MK4004GAH (1.8″ hard disk)	K9K8G08U0M (NAND flash)
Rotational speed (rpm)		4,200	N/A
Avg. rotation time (ms)		7.1	N/A
Avg. seek time (ms)		15	N/A
Latency (512 B)	Read	22.1 (ms)	25 (μs)
	Write	22.1 (ms)	200 (μs)
	Erase	N/A	1.5 (ms)
Power (mW)	Active	1400	33
	Idle	400	0.13
	Standby	200	N/A

transfer time of the disk is invisible but we modeled it as 7 us per sector in our simulator.

In experiments, we assigned the capacity of the flash memory to 4, 8, 16, 64, 128, and 256 MB. Our flash memory simulator simulates a NAND flash memory and its controller. It includes an FTL, which consists of an address mapper, a reclamation manager, and a wear leveler. We have implemented a page-level address mapping policy [22] and adopted a greedy policy for block reclamation and wear leveling. At each I/O access, when writes to the NVC occur, the FTL of the flash memory will check whether there is enough space (i.e., block) to write data. If the number of free blocks reaches the lower bound, RECYCLING_THRESHOLD, GC will starts. Otherwise, the FTL will just tries to get a free block and write data on it without invoking GC. In this work, this threshold value is set to 2.

We simulate LRU and PAW to manage an NVC and maintain *lru_list* in each cache replacement algorithm at a DRAM (refer to Fig. 3 for this list). We also keep a table for an address mapper in an FTL within the DRAM. Our metrics are an average I/O response time per block for performance, total energy consumption, a flash write count, a flash erase count, and a cache hit rate. To evaluate LRU and PAW, we employed a generic application I/O trace [20], which was collected on a Windows XP platform while several well-known Windows applications such as Powerpoint, Internet Explorer, and Outlook Express are running. Table 2 shows the trace information and collection environment. The request size of writes amounts to 1,288 MB and the fraction of writes is found to be 73% of the total request size. We analyzed the access patterns of reads and writes using a sequentiality threshold and the result is shown in Table 3. We found that the fraction of random requests is higher than that of sequential ones among all the requests, which are found to be 63% and 37%, respectively.

At each block access, PAW determines its access pattern. To determine this, it utilizes a sequentiality threshold. In simulations, we set the sequentiality threshold to 32 blocks (or sectors). Every I/O is requested with a start LBA and an LBA count, and thus if the count is larger than 32, all the LBAs belonging to this request are labeled sequential. We focused on the LBA number because two continuous different requests can be merged into a large sequential one by applying I/O clustering-based queuing. This method is simple but effective and thus many researchers are found to employ it in the literature including [29]. The reason why the threshold 32 is selected is that Windows XP allocates data to each file in the unit of a page and 4 pages (i.e., 32 sectors) are most frequent. Since our trace came from Windows XP, we thought that 32 sectors can be the boundary of sequential and non-sequential accesses.

Table 2 Collection information and trace information [20].

Collection environment	OS	Microsoft Windows XP
	RAM	512 MB
	Disk	8 GB
Trace information	Total execution time	3 hr 30 min
	Request count (r/w)	72822 (31208/41614)
	Request size (r/w)	1772 MB (484/1288)
	Working-set size	955 MB

Table 3 Trace pattern analysis.

Pattern			Count (Percentage)		Total
Requests	Sequential	Read	26783	12375	72822
		Write	(37%)	14408	
	Random	Read	46039	18833	
		Write	(63%)	27206	
Sectors	Sequential	Read	3218520	828812	3631532
		Write	(89%)	2389708	
	Random	Read	413012	163617	
		Write	(11%)	249395	

In addition, we adopt I/O clustering to the requests in the queue of a device driver. As a result of mergence among sequential requests, if a request continues to grow large, that is, if an LBA of a currently accessed block is continued to LBAs of prior accessed blocks, the block will have high probability of being labeled as sequential. Otherwise, it is labeled random.

5.2 Simulation Results

We demonstrated how our approach can achieve significantly better I/O performance than a traditional one while enhancing the sequentiality of requests in the queue through extensive simulations. We also show how much it can reduce the number of writes and thus decrease erases of a flash memory used as an NVC. Varying the NVC size, we evaluated 4 combinatory pairs of two cache algorithms and two queuing methods for the generic trace shown in Table 2 and 3, where the former consists of LRU and PAW and the latter consists of simple queuing and I/O clustering with time-bounded plugging. The default queue size (*MAX_REQ_QUEUE_SIZE* in Fig. 3) is set to 5 and the timeout value to unplug requests in the queue (*PLUGGING_TIMEOUT* in Fig. 3) is 60 ms.

5.2.1 Sequentiality

Table 3 shows the analysis of simulation results for the used trace. This is the case that LRU and a simple queue are used. We show the number of sequential and random requests, which may be writes or reads. We also show the

accumulated number of sectors for classified requests. At the process of each request within the trace, our simulator checked if a block belonging to each request is sequential or not. The simulator also determined whether it is a write or a read access. We notice that sequential writes amounts to 2389708 in terms of sectors, which is about 66% of all the processed sectors. This means that writes with sequentiality should be treated properly for the overall system performance. Table 4 shows the analysis of simulation results for the case that I/O clustering based queuing is used.

As is shown in Tables 3 and 4, after I/O clustering is applied the number of sequential requests was reduced from 26783 to 26620 while the total request number was shrunk from 72822 to 72659 by the same amount. This means that some of the requests were merged into large sequential ones by I/O clustering and thus the number of sequential requests was decreased by 163. That is, the PAW algorithm can strengthen the sequentiality which may reside between consecutive I/O requests. Therefore, our approach will be beneficial in boosting the overall disk performance since a disk has faster processing time with sequential requests due to much shorter seek time and rotational delay than with random ones.

Table 5 shows detailed experimental data of a disk for simple queuing and I/O clustering when LRU is used with a 4 MB NVC. Here, LRU is used to manage an NVC and simple queuing is used to manage a queue by a queue manager in the device deriver in Fig. 2. Each column shows our measured values, which are accumulated values of queue delay, disk transition time, seek time, transfer time, and rotational delay while our simulator executes a generic trace. The unit is ms. Although the sequentiality was improved very little in Table 3 and 4, the queue delay of the disk was found to be larger when a simple queue was used. We found that the queue delay of simple queuing is about 6.3 times larger than that of I/O clustering-based queuing.

We got the result of improved average I/O response times due to the shortened queue delay when I/O clustering-based queuing was used. The reason why the queue delay for simple queuing was increased in comparison with I/O clustering-based queuing is that simple queuing has no time out mechanism and only starts processing all the requests in the queue when the number of accumulated requests reaches the limit. On the other hand, I/O clustering-based queuing has time-bounded plugging and we described this in Fig. 3 and Sect. 4.2 in the paper. In the mean time, the number of merged I/O requests is highly related to the time-bounded plugging. The shorter the time out is, the less I/O clustering will occur. Thus, deciding the time out value in I/O clustering is important. In our paper, we set this value to 60 ms.

Table 4 Trace pattern after I/O clustering-based queuing is applied.

Pattern			Count		Total
Requests	Sequential	Read	26620	12368	72659
		Write		14252	
	Random	Read	46039	18833	
		Write		27206	

Table 5 Comparison of measured disk parameters for simple queuing and I/O clustering (unit: ms).

	qdelay	diskdelay	seektime	xfertime	rotational_delay
simple queuing	22132781	1100500	266570	23895	234976
I/O clustering	3520006	1034900	265799	23893	234254

5.2.2 Average I/O Response Time

Figure 4 shows the average I/O response times of 4 combinatory pairs of cache management and queuing method for a generic trace when the size of an NVC varies. We can notice that PAW + I/O clustering shows the fastest I/O response time as the cache size grows among all the pairs. LRU + I/O clustering has shorter I/O response time than PAW + I/O clustering till 16 MB of the NVC, but after 16 MB PAW + I/O clustering has shorter response time than LRU + I/O clustering by up to 77%. This comes from that when an NVC is small many random requests are likely to be evicted from it due to its limited capacity and thus PAW has few chances to keep more random, that is, likely to be temporal, blocks in the NVC. As a result, with PAW, evicted random blocks are supposed to have worse influence on a disk's behaviors due to an irregular mix of sequential and random accesses than with LRU. Other observations are two-fold. First is that PAW has wholly better performance than LRU with the help of an I/O clustering-based queuing method. Second is that the average I/O response time looks convex as the cache size increases.

The average I/O response time was found to be proportional to the flash erase count. In case of LRU, if the NVC size is small there are many requests forwarded to a disk due to cache misses irrespective of access patterns. However, since, as the NVC size grows, the NVC absorbs many reads and writes requests, writes to the NVC increase and thus the count of erase in the NVC increases. This phenomenon occurred until the NVC size reached 64 MB and appeared weak beyond this value.

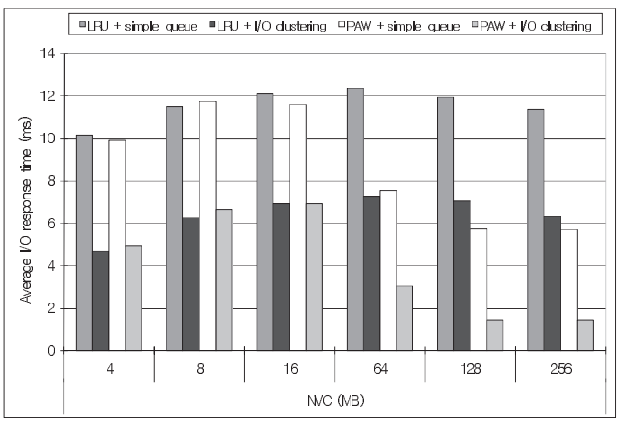

Fig. 4 Average I/O response times of 4 combinatory pairs of cache management and queuing method for a generic trace.

5.2.3 Flash Write Count and Erase Count

Figures 5 and 6 show the flash write counts and erase counts of 4 combinatory pairs of cache management and queuing method for a generic trace, respectively. As we can notice in Fig. 5, the write block count of an NVC was improved by about 86% over LRU when PAW was employed. This is almost consistent with the 89% percentage of sequential I/O blocks in Table 3. For this difference between two percentages, since all the block accesses go through the NVC when LRU is used, but only random blocks and sequential blocks with temporal locality such as loop data are forwarded to the NVC with PAW, the fraction of blocks accessed to the NVC with PAW is found to decrease by 3%.

Basically, flash writes come from both types of read and write accesses. For reads, when a miss occurs, a flash write is done. For writes, flash writes are generated when a cache hit as well as a cache miss occur. When a cache hit occurs, the target page should be invalidated and its data should be moved (written) to a new page. When a cache miss occurs, data is written to a new page only. Thus, in case that a cache hit occurs from a write access, double pages are consumed in comparison with a cache miss from a read access and more erases will be done.

From our observations, as the NVC size grows, cache hits from write accesses increased. Thus, erases from cache hits increased very much as is shown in Fig. 7. In Fig. 7, when the NVC size grows, the flash erase count from cache misses also increased. This is because as the NVC grows it absorbs read and write blocks, which are random. Due to these occupied blocks, most sequential writes will have a few chances to stay longer in the NVC relatively. Thus, many write accesses will be evicted from the NVC, causing GCs only. Consequently, although the flash write count varies little, the flash erase count increased.

In Fig. 5 and 6, the increment of writes to a flash memory is found to be smaller than that of erases in a flash as the NVC size increases with LRU and PAW. The reason is that, as the NVC grows writes to a flash memory can be ascribed to write requests coming from a trace rather than writes to an NVC due to cache misses with both LRU and PAW. PAW has relatively fewer writes to the NVC due to directing most of sequential requests including many writes to the disk over LRU and has more chances of avoiding increase of the erase count of a flash memory. Thus, we notice that erases in a flash memory with PAW are reduced by 84% to 100% (this means there being no GCs with PAW at 256 MB) over LRU in Fig. 6.

For PAW, a peak erase count, which resulted in the worst average I/O response time, appeared at about 16 MB, where the NVC size is rather small than that for LRU. This comes from that PAW enables to store a larger number of random data to the NVC than LRU and evict sequential data in the NVC better.

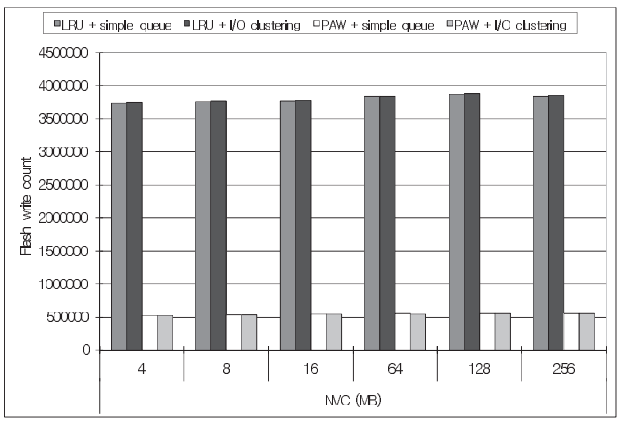

Fig. 5 Flash write counts of 4 combinatory pairs of cache management and queuing method for a generic trace.

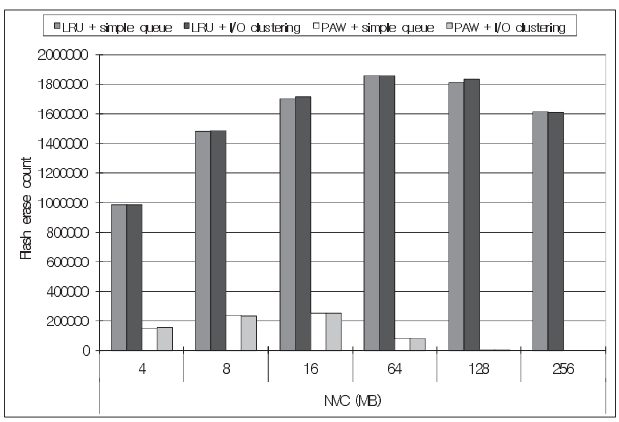

Fig. 6 Flash erase counts of 4 combinatory pairs of cache management and queuing method for a generic trace.

5.2.4 Energy Consumption

Figure 8 shows the total energy consumption of the storage system for 4 combinatory pairs of cache management and queuing method for a generic trace. We found that PAW-combined approaches have good energy saving over LRU-combined ones. This is because LRU-combined approaches

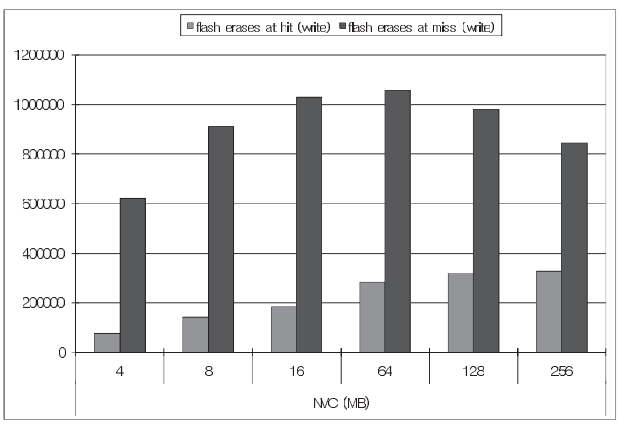

Fig. 7 Flash erase counts at NVC hit and miss for write accesses.

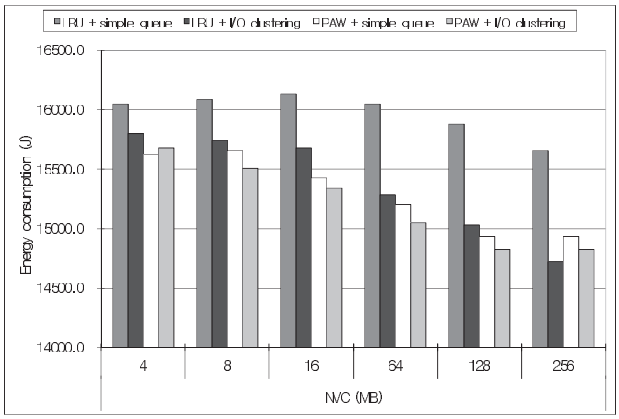

Fig. 8 Energy consumption of 4 combinatory pairs of cache management and queuing method for a generic trace.

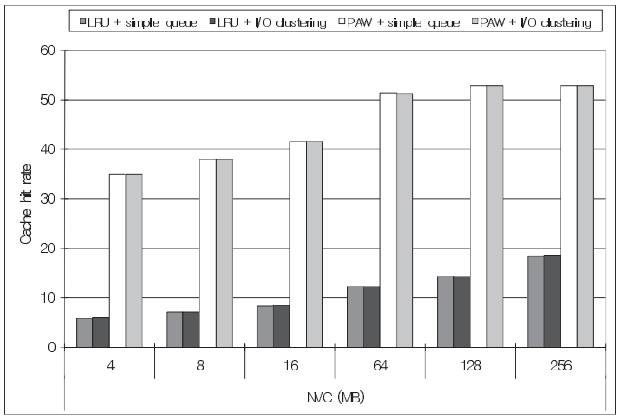

Fig. 9 Cache hit rates of 4 combinatory pairs of cache management and queuing method for a generic trace.

are affected by a mix of sequential and random data, evicting many I/O blocks to a disk and spinning it on and on. In addition, PAW-combined approaches have only sequential requests directed to a disk and random data maintained in a flash memory, taking less spinning time and reduced seek time of the disk. At 256 MB in Fig. 8, the energy consumption of LRU + I/O clustering appears lowest. Our observation from experimental data is that LRU + I/O clustering saves much energy by storing data to an NVC and PAW-combined approaches lose some chances of energy saving since PAW still forwards many sequential requests to a disk, keeping it spinning in an active mode and thus consuming a lot total energy although the energy consumption of an NVC is small in comparison with LRU.

5.2.5 Cache Hit Rate

In Fig. 9, PAW has at least 2.8 times higher cache hit rates than LRU regardless of queuing methods. This is because PAW directs random data, which has high probability to have temporal locality, to an NVC and sequential data to a disk, and it can maintain more random data within the NVC as the NVC grows. The reason why I/O clustering-based queuing is ineffective to raising the NVC hit rate is ascribed to the fact that a cache hit rate is only affected by the volume of random data forwarded to an NVC and its size. In other words, the requests merged by I/O clustering are likely to have sequentiality and most of them will be transferred to a disk without any accesses to the NVC with PAW. Consequently, the hit rate of the NVC will hardly be influenced by I/O clustering as shown in Fig. 9.

6. Related Work

There have been a lot of researches on enhancing the overall I/O performance in disk-based storage systems. Representatives are cache management and queue scheduling techniques. Various caching algorithms in a lot of literature have been studied until now. For example, [2] tried to optimize the system performance by renovating LRU at the page cache level. [1], [6]–[9] investigated a write cache at the level of a device controller while recent work such as [13], [14], [20], [23] studied the usage of a flash memory as an NVC in a disk-based storage system.

In disk array-based servers using data striping mechanisms such as RAID 5, since updates of data as well as parities have significant influence on the overall I/O performance, many techniques employing an NVC have been studied [1], [5]–[8]. These works mainly focused on developing destage algorithms, which take the role of evicting the least valuable dirty blocks from the NVC effectively if there is no free space in the NVC when a new write arrives at a storage controller.

In mobile systems, studies of using an NVC as a write cache have appeared with the purpose of reducing write latency [9]. Recently, adoption of a NAND flash memory as a write cache has been observed frequently. Samsung and Microsoft developed a hybrid hard disk drive technology, which combines a hard disk with a NAND flash memory as an NVC, to boost the performance, reduce the power consumption, and increase the reliability of mobile computers [10]–[12]. Bisson et al. tried to improve a hybrid hard disk drive technology with several supports from I/O subsystems including I/O redirection techniques and polished NVC techniques in terms of performance and energy [13], [14].

At a device controller level, queue scheduling techniques have been researched a lot such as first-come-first-serve (FCFS), shortest seek time first (SSTF), SCAN, and cyclic SCAN (CSCAN) [24], but there have been scarce work on combining a cache algorithm and a queuing technique. [1] proposed WOW which is a write request ordering technique a non-volatile cache, combining a cache eviction policy and a queue scheduling method, aiming at achieving temporal locality and spatial locality together synergistically. However, WOW does not consider the actual I/O latency of an NVC since it assumes to employ a volatile DDR memory as an NVC.

Unlike prior work, ours focuses on combining a cache algorithm and a queue scheduling method at two levels of a

disk-based storage system, which are a device driver of an OS and a device controller of a storage device, while considering the real I/O features of a NAND flash memory for an NVC. Our work is designed to face the characteristics of mobile workload patterns and aims at enhancing the overall I/O performance from the synergistic effect of combining a pattern-aware write cache policy and an I/O clustering-based queuing method to strengthen the sequentiality.

7. Conclusions

We proposed a performance-enhanced write cache policy for a NAND flash memory in disk-based mobile storage systems. The proposed policy redirects non-sequential requests to an NVC and sequential ones to a disk in order to reduce the overall system I/O latency. To strengthen the sequential operations in a disk we adopted an I/O clustering-based queue scheduling algorithm. To evaluate the proposed approach, we have built a practical hard disk simulator with an NVC of a NAND flash memory. Extensive simulations showed that our policy significantly improves the overall system I/O performance by reducing the overhead from a non-volatile cache considerably over a traditional one, achieving a high efficiency in energy consumption.

As future work, we plan to study comprehensive integrated I/O performance performance-optimizing techniques. To the best of our knowledge, there has been no investigation of the overall system I/O performance, which considers a page cache in a file system and a request queue in a device driver, and a cache and a queue in a device controller concurrently. We also plan to investigate energy-optimizing techniques for a disk-based mobile storage device with an NVC.

Acknowledgments

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-C1090-1031-0004), and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.2010-0004980). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.20100018873).

References

[1] B.S. Gill and D.S. Modha, "WOW: Wise ordering for writes - combining spatial and temporal locality in non-volatile caches," Proc. 4th USENIX Conference on File and Storage Technologies (FAST), San Francisco, CA, Dec. 2005.

[2] N. Megiddo and D.S. Modha, "ARC: A self-tuning, low overhead replacement cache," Proc. 2nd USENIX Conference on File and Storage Technologies (FAST), March-April 2003.

[3] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," Proc. Winter 1993 USENIX Conference, pp.405–420, San Diego, CA, Jan. 1993.

[4] F. Douglis, F. Kaashoek, K. Li, R. Caceres, B. Marsh, and J.A. Tauber, "Storage alternatives for mobile computers," Proc. 1st Symposium on Operating Systems Design and Implementation (OSDI), 1994.

[5] J. Menon and J. Cortney, "The architecture of a fault-tolerant cached RAID controller," Proc. 20th Annual International Symposium on Computer Architecture (ISCA), pp.76–86, 1993.

[6] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," IEEE Trans. Comput., vol.47, no.2, pp.228–235, 1998.

[7] Y.J. Nam and C. Park, "An adaptive high-low water mark destage algorithm for cached RAID5," Proc. 2002 Pacific Rim International Symposium on Dependable Computing (PRDC), Dec. 2002.

[8] M. Alonso and V. Santonja, "A new destage algorithm for disk cache: DOME," Proc. 25th Euromicro Conference (EUROMICRO), 1999.

[9] P. Biswas and K.K. Ramakrishnan, "Trace driven analysis of write caching policies for disks," Proc. 1993 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 1993.

[10] Microsoft, ReadyDrive and Hybrid Disk. http://www.microsoft.com/whdc/device/storage/hybrid.mspx

[11] Samsung, Hybrid Hard Disk Drive. http://www.samsung.com/Products/HardDiskDrive/news/HardDiskDrive_20050425_0000117556.htm

[12] R. Panabaker, "Hybrid hard disk & ReadyDrive™ technology: Improving performance and power for Windows Vista mobile PCs," Proc. Microsoft WinHEC 2006. http://www.microsoft.com/whdc/winhec/pres06.mspx

[13] T. Bisson and S. Brandt, "Reducing hybrid disk write latency with flash-backed IO requests," Proc. 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2007.

[14] T. Bisson, S. Brandt, and D. Long, "A hybrid disk-aware spin-down algorithm with I/O subsystem support," Proc. 26th IEEE International Performance Computing and Communications Conference (IPCCC), New Orleans, Louisiana, USA, April 2007.

[15] G. Lawton, "Improved flash memory grows in popularity," Computer, vol.39, no.1, pp.16–18, Jan. 2006.

[16] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," Proc. USENIX Winter Technical Conference, pp.155–164, 1995.

[17] Intel Corporation, Understanding the flash translation layer (ftl) specification. http://www.intel.com/design/flcomp/applnots/297816.htm

[18] E. Shriver, Performance modeling for realistic storage devices, Ph.D. dissertation paper, Department of Computer Science, New York University, May 1997.

[19] D.P. Bovet and M. Cesati, Understanding the linux kernel, O'Reilly, 3rd ed., 2005.

[20] Y.-J. Kim, S.-J. Lee, K. Zhang, and J. Kim, "I/O performance optimization technique for hybrid hard disk-based mobile consumer devices," IEEE Trans. Consum. Electron., vol.53, no.4, pp.1469–1476, Nov. 2007.

[21] Samsung, NAND flash memory. http://www.samsung.com/products/semiconductor/NANDFlash/SLC_LargeBlock/8Gbit/K9K8G08U0M/K9K8G08U0M.htm

[22] T.-S. Chung, S. Park, M.-J. Jung, and B. Kim, "STAFF: state transition applied fast flash translation layer," Lect. Notes Comput. Sci. (LNCS) 2981, Springer-Verlag, 2004.

[23] P. Biswas, K.K. Ramakrishnan, and D. Towsley, "Exploiting non-volatile memory in disks for write caching," Technical Report, UM-CS-1994-067, University of Massachusetts, Amherst, Sept. 1994.

[24] B.L. Worthington, G.R. Ganger, and Y.N. Patt, "Scheduling algo-

rithms for modern disk drives," Proc. 1994 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp.241–251, 1994.
[25] Toshiba, MK4004GAH. http://www3.toshiba.co.jp/storage/english/spec/hdd/mk4004gs.htm
[26] F. Chen, S. Jiang, and X. Zhang, "SmartSaver: Turning flash drive into a disk energy saver for mobile computers," Proc. I 11th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'06), Tegernsee, Germany, Oct. 2006.
[27] Y.-J. Kim, K.-T. Kwon, and J. Kim, "Energy-efficient file placement techniques for heterogeneous mobile storage systems," Proc. 6th ACM & IEEE Conference on Embedded Software (EMSOFT), Seoul, Korea, Oct. 2006.
[28] Y.-J. Kim and J. Kim, "Device-aware cache replacement algorithm for heterogeneous mobile storage devices," Proc. 3rd International Conference on Embedded Software and Systems (ICESS), Daegu, Korea, May, 2007, Lect. Notes Comput. Sci. (LNCS), 4523, pp.13–24, May 2007.
[29] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references," Proc. 4th Symp. Operating System Design and Implementation, pp.119–134, San Diego, California, USA, Oct. 2000.

Young-Jin Kim received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, Korea, in 1997, 1999, and 2008, respectively. From 1999 to 2003, he was with Electronics and Telecommunications Research Institute (ETRI), Daejoen, Korea. He is currently an assistant professor in the Department of Computer Science and Engineering, Sun Moon University, Asan, Korea. His research interests include embedded systems and software, mobile storage systems and software, and power measurement and analysis.

Jihong Kim received the B.S. degree in computer science and statistics from Seoul National University, Seoul, Korea, in 1986, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle, WA, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a Professor in the School of Computer Science and Engineering, Seoul National University, Seoul, Korea. His research interests include embedded software, low-power systems, computer architecture, and multimedia and real-time systems.

Jeong-Bae Lee received the B.S. and M.S. degrees from Kyongpook National University. He got his Ph.D. degree from Hanyang University in 1995. From 1982 to 1991, he was a Senior Researcher in Electronics and Telecommunications Research Institute (ETRI), Daejoen, Korea. He was a visiting professor in U.C. Irvine from 1996 to 1997. He is now a Professor in Department of Computer Science and Engineering, Sun Moon University, Asan, Korea. His research interests include embedded systems, real-time systems, and real-time communication protocols.

Kee-Wook Rim received the B.S. degree in electronic engineering from Inha University in 1977 and the M.S. degree in computer science from Hanyang University in 1986. He got his Ph.D. degree in computer science from Inha University in 1994. From 2001 to 2003, he was the President of Institute of Computer and Software Research in Electronics and Telecommunications Research Institute (ETRI), Daejoen, Korea. He is now a Professor in Department of Computer Science and Engineering, Sun Moon University, Asan, Korea. His research interests include computer networks, embedded systems, and database systems.