PAPER

# DAC: A Device-Aware Cache Management Algorithm for Heterogeneous Mobile Storage Systems

**Young-Jin KIM**[†], ***Member and*** **Jihong KIM**[††a)], ***Nonmember***

**SUMMARY** In recent years, heterogeneous devices have been employed frequently in mobile storage systems because a combination of such devices can supply a synergistically useful storage solution by taking advantage of each device. One important design constraint in heterogeneous storage systems is to mitigate I/O performance degradation stemming from the difference between access times of different devices. To this end, there has not been much work to devise proper buffer cache management algorithms. This paper presents a novel buffer cache management algorithm which considers both I/O cost per device and workload patterns in mobile computing systems with a heterogeneous storage pair of a hard disk and a NAND flash memory. In order to minimize the total I/O cost under varying workload patterns, the proposed algorithm employs a dynamic cache partitioning technique over different devices and manages each partition according to request patterns and I/O types along with the temporal locality. Trace-based simulations show that the proposed algorithm reduces the total I/O cost and flash write count significantly over the existing buffer cache algorithms on typical mobile traces.

***key words:*** *heterogeneous mobile storage, performance optimization, device-aware cache management, dynamic cache partitioning, workload-aware management*

## 1. Introduction

Mobile and ubiquitous computing systems such as PDAs, PMPs, and MP3 players have become extremely popular among consumers. Under these computing environments, it is required that a large volume of data should be stored and processed fast and reliably because end-users want to use high-performance and high I/O load applications such as video on demand, games, and multimedia players. In recent years, various non-volatile device technologies such as FeRAM, PRAM, MRAM, and micro-electro mechanical system (MEMS) besides NAND flash memory have been developed in order to meet such demands with reasonable trade-offs across performance, energy consumption, and price, targeting hard disk drives, which are the most widely-used secondary storage devices [1], [2]. Among them, NAND flash memory is recognized to be most commercially available and a competitive alternative to hard disk drives as a secondary storage device [3]. The NAND flash memory technology has been evolved into two types: single-level cell (SLC) NAND and multi-level cell (MLC) NAND [4].

### 1.1 Use of Heterogeneous Storage Devices

As is known well, hard disks have attractively low cost per bit, but they are known as significant power consumers and show poor performance for random I/O requests. The in-coming storage devices including NAND flash memory have relatively fast response times, very low power consumptions, and strong shock resistance, but their costs per bit are much higher than hard disks. For example, as shown in Table 1, SLC NAND flash memory has at least about 6 times higher cost per GB than small form-factor disk drives these days. However, SLC NAND flash memory shows at least 42 times lower power consumption in the active mode and 95 times faster write response time than hard disks. Flash memory also should be erased before write and has a limited write/erase cycles, though this is invisible in Table 1. The fast response time of NAND flash memory and the low cost per bit of hard disks can be attractive in implementing mobile storage systems, but the high power consumption of hard disks and high cost per bit of flash memory may be problematic. Therefore, using such devices together is expected to supply a synergistically useful storage solution by taking advantage of each device. This is the reason why various combinations of different storage devices have emerged frequently in mobile storage systems.

There are many researches to combine heterogeneous storage devices in academy as well as industry. A representative example is using a hard disk and a NAND flash memory (hereafter, a flash or a NAND flash mean a SLC NAND flash unless described otherwise). Combinations of a hard disk and a flash memory have been widely studied and they can be categorized into two groups: one uses a flash memory as a non-volatile cache [5]–[13] and the other uses a flash

**Table 1** Characteristics of typical small form-factor hard disks and NAND flash memories [4], [14], [20], [21].

| Device | | Hard disk | | NAND flash | |
|---|---|---|---|---|---|
| | | 2.5″ | 1.8″ | SLC | MLC |
| Latency (512B) | Read | 19.1 (ms) | 22.1 (ms) | 25 (us) | 60 (us) |
| | Write | 19.1 (ms) | 22.1 (ms) | 200 (us) | 800 (us) |
| | Erase | N/A | N/A | 1.5 (ms) | 1.5 (ms) |
| Power (mW) | Active | 2300 | 1400 | 33 | > 33 |
| | Idle | 950 | 400 | 0.13 | > 0.13 |
| | Standby | 250 | 200 | N/A | N/A |
| Cost per GB ($) | | 0.82 | 1.82 | 10.89 | 3.34 |

Manuscript received January 9, 2008.
Manuscript revised July 20, 2008.

[†]The author is now with the Department of Computer Science and Engineering, Sun Moon 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.
a) E-mail: jihong@davinci.snu.ac.kr

DOI: 10.1093/ietisy/e91–d.12.2818

memory as a secondary storage device [14], [15]. Both types aim at the overall performance enhancement or energy saving. As an another example, combined uses of a hard disk and MEMS were researched to achieve I/O performance improvement by using MEMS-based storage as a write buffer or a secondary storage device [16], [17]. Most recently, hybrid flash memory solutions using MLC and SLC NAND flash memory devices concurrently has appeared in order to take advantage of high speed and long erase cycles of the SLC NAND flash as well as large capacity and low production cost of the MLC NAND flash [18], [19]. These plentiful examples, we believe, reflect that the effectiveness and benefit derived from heterogeneous storage solutions would be significant if separate devices would be exploited together harmoniously.

### 1.2 Design Constraints of Heterogeneous Storage Systems

As mentioned above, there are several design constraints to be considered in combining heterogeneous storage devices into an integrated mobile storage system. Energy consumption, performance, and reliability may be counted as three major constraints other than cost. In most mobile computing systems, power consumptions of individual devices are important since they have significant effects on the lifetime of the whole system. On the other hand, the overall I/O performance fluctuates according to workloads and load distribution due to different access times of heterogeneous devices. Therefore, performance is another critical constraint in implementing heterogeneous storage devices. Finally, each device has different ruggedness and a mean time before failure (MTBF). Operating only one device during a large amount of the whole operation time is likely to give rise to low reliability if the device would have weak ruggedness or a short MTBF.

In mobile storage devices, finding an optimal solution under these three design constraints is an important research problem. However, the overall design space becomes too big to be effectively addressed in this paper when all these design requirements are simultaneously considered. In order to deal with this complex optimization issue, in this paper, we concentrate on the performance optimization in exploiting heterogeneous storage devices. In order to take a step-by-step approach, we already researched energy-efficient heterogeneous mobile storage management, which analyzed mobile workloads and proposed file migration techniques based on them to spin down a hard disk for high energy saving [14]. Exploring a larger design space for optimal design parameters of both energy and performance will be, we believe, a major future work item.

### 1.3 Performance Optimization Techniques for Heterogeneous Storage Systems

Let us assume that a heterogeneous storage system consists of a device part and a host part: the former is composed of heterogeneous devices and a device controller, which has a device cache and a request queue, and controls these components. The latter is composed of an operating system including a task scheduler, a file system, a buffer cache algorithm, and a device driver. There are abundant performance optimization techniques for heterogeneous storage systems similarly to homogeneous storage ones. As hardware-based techniques, device cache management and/or request queue scheduling techniques are available at the device controller level. Software-based techniques consist of compiler-aware and operating system aware approaches. Transforming codes for device I/O scheduling belongs to the compiler-aware approach. Task I/O scheduling in the task scheduler, request queue scheduling in the device driver, device I/O scheduling in the file system, and buffer caching algorithms are classified into the operating system aware approach. In this work, we bound our interest to a buffer caching algorithm appropriate for heterogeneous storage systems.

## 2. Buffer Caching Algorithms for Heterogeneity

In a lot of literature, various buffer caching algorithms at the level of operating systems in various storage systems have enabled significant performance enhancement by filtering I/O requests directed to devices over a widely-used buffer caching algorithm, LRU [22]. However, few buffer caching algorithms for heterogeneous storage systems have been reported. Therefore, in order to meet newly-emerging hybrid storage technology trends as aforementioned, it is highly required to investigate and develop well-devised buffer caching algorithms appropriate for the combinations of heterogeneous storage devices. Towards this end, we need to concretize what properties are required to cope with heterogeneity in buffer caching algorithms. In this section, we look into requirements in buffer cache algorithms for heterogeneity and digest such requirements by explicit descriptions.

### 2.1 Buffer Caching Algorithms and Work Balance

In multi-device based storage systems regardless of their being homogeneous or heterogeneous, work balance is a crucial problem. *Work (or load) balance* means that work is evenly spread over the available storage devices and thus each work done by each device becomes equal [23], [24]. Work balance depends on I/O request distribution over devices. And the I/O request distribution depends on buffer cache management directly. This is because a buffer cache manages blocks based on its management policy and thus has a filtering effect on which device the requested blocks should be forwarded to and how many they should be. Hence, different buffer cache management algorithms will produce different I/O request distributions. In homogeneous storage systems, uniform I/O accesses are likely to cause work balance and good work balance will enable the overall I/O performance to approach a minimum.

However, in heterogeneous storage systems, the situation is different. In heterogeneous storage systems, work balance is fragile due to a large gap between access times of different devices. Therefore, buffer caching algorithms for heterogeneous storage devices should make more elaborate efforts in order to achieve work balance effectively in that they should take into accounts different access times of different devices as well as workload patterns. That is, *device awareness* in buffer cache algorithms should be provided in heterogeneous storage systems. In detail, *device-aware (or cost-aware)* buffer caching algorithms need to manage each cache block by taking accounts of the access time of the device, which the block would be serviced from if a cache miss would occur, together with its temporal locality. However, existing buffer caching algorithms like LRU and LRU-variants in traditional operating systems are not aware of different access times of heterogeneous devices and thus treat all cache blocks as if they had the same miss penalties from different devices. Therefore, using LRU-like algorithms will make it more difficult to achieve work balance over heterogeneous storage systems.

There are a few prior device-aware buffer caching algorithms seeking to achieve work balance over heterogeneous devices. [23] and [25] investigated device-aware buffer cache management algorithms using different costs on heterogeneous disks and a storage pair of a disk and a NAND flash memory, respectively. Work balance, however, is not achievable always in heterogeneous systems. For example, highly skewed sequential I/O accesses over a slower device (e.g., a hard disk) can derive workload imbalance despite the buffering effect of a buffer cache (such phenomenon was observed to occur often in heterogeneous storage systems, especially with a large gap in access times between different devices like a pair of a disk and a NAND flash memory). This suggests that we should try to minimize the total I/O cost rather than to accomplish work balance. Thus, the proposed device-aware caching algorithm is designed to aim at optimizing the overall I/O performance rather than achieving work balance.

### 2.2 Device Awareness in Buffer Caching Algorithms

Device awareness in the buffer cache management can be supported in two ways. One is to deal with blocks with heterogeneous access times within a single partition cache. This approach assigns different values, which correspond to different access times of heterogeneous devices, to all the blocks within the buffer cache and evicts the block with the least value among all the blocks when a cache miss occurs. The GreedyDual algorithm [26], which has been widely used in the Web cache communities, operates similarly to this approach. GreedyDual combines locality and miss penalty concerns to achieve a good overall performance, but it cannot be used for a buffer cache without modifications because it usually deals with a file as a unit of caching. Furthermore, GreedyDual itself was not designed to be device-aware across different storage devices. Compared with GreedyDual, our work targets the buffer cache management on heterogeneous storage devices and deals with a block as a unit of caching.

The other approach is to partition a buffer cache into separate partitions (one per device) and managing them according to the assigned value to each block. This approach should take into accounts appropriately the determination of partitioning sizes as well as the management of cache blocks within each partition. There are two partitioning approaches: static partitioning and dynamic partitioning. The static partitioning approach is simple but cannot adapt workload variations, compared with the dynamic partitioning approach. Our work belongs to the dynamic partitioning approach, taking accounts of heterogeneity in access times of different devices and varying workload patterns.

Our partitioning technique is not the first dynamic cache partitioning approach. [23] and [25] employed dynamic cache adjusting techniques, but they just tried to mitigate work imbalance. In comparison with these works, our contributions are described as follows.

1) Aiming at optimizing the total I/O cost rather than balancing work over devices, we propose a novel buffer cache management algorithm for heterogeneous systems based on a dynamic cache partitioning technique. This technique divides the buffer cache into partitions of the corresponding devices, depending on different miss penalties and varying workload patterns.

2) For the management of blocks within each partition, we introduce a cache management algorithm which combines the temporal locality, sequentiality, and I/O types. This technique augments the GreedyDual algorithm, but is re-designed to utilize the individual characteristic of each device (e.g., sequential and read blocks may be favorable for a disk and a flash memory in terms of performance and wear-leveling) and operate in harmony with the dynamic cache partitioning technique.

3) Through extensive trace-driven simulations, we show the effectiveness of the proposed buffer caching algorithm for a heterogeneous storage pair of a hard disk and a NAND flash memory. However, the proposed technique can be extended to other heterogeneous storage systems. For instance, our approach can be applied to a pair of MLC and SLC NAND flash memories. This is because an MLC flash consumes more power and longer access time than an SLC flash as a hard disk does than an SLC flash. In this paper, we mainly focus on a heterogeneous storage pair of a hard disk and a NAND flash memory due to the space limit of the paper.

The rest of the paper is organized as follows. We present a motivational example in Sect. 3. The proposed device-aware cache management algorithm is described in Sect. 4. The experimental results are discussed in Sect. 5. Section 6 explains related work and Sect. 7 concludes with a summary.

## 3. Motivational Example

As mentioned in the previous section, if a storage system consists of heterogeneous storage devices, a buffer cache

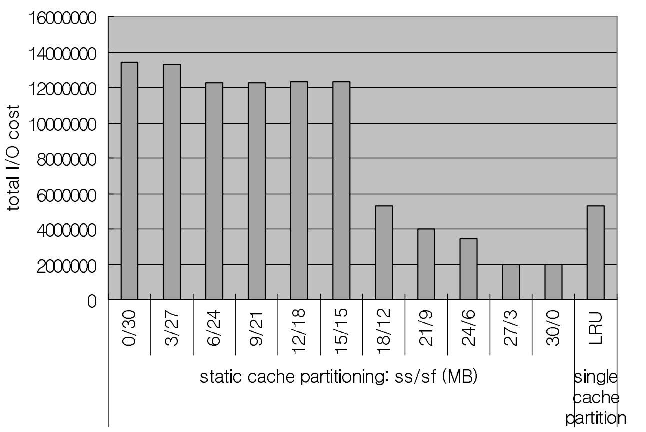


**Fig. 1** Total I/O cost in a heterogeneous storage system with a 30 MB cache for a real mobile workload from [25]. As the cache partition size for a disk grows, the total I/O cost decreases largely but has the smallest at a cache partition configuration of 27 MB for a disk and 3 MB for a flash memory, outperforming LRU with a single cache partition by about 63%.

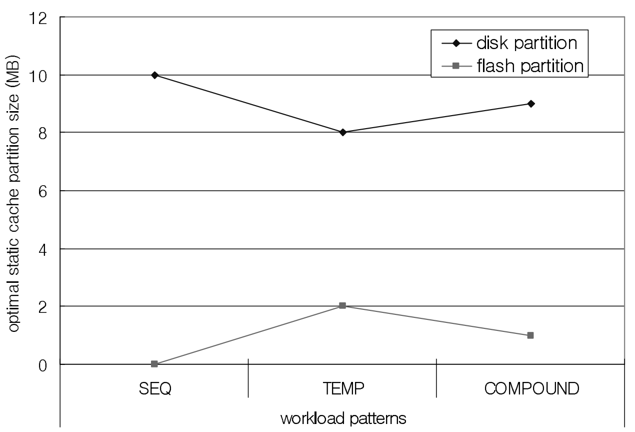


**Fig. 2** Comparisons of the optimal static cache partition sizes in terms of the total I/O cost for three different synthetic mobile workloads in a storage system with a disk and a NAND flash memory. We notice that different workload patterns derive different best static cache partition configurations, that is, static cache partitioning policies.

management algorithm needs to be device-aware. In this section, we emphasize this motivation using a small set of experimental results. In particular, we demonstrate two observations. First, we show that a simple static cache partitioning technique can outperform a single partition cache algorithm. Second, we compare the I/O performances of the static partitioning techniques for different workloads, and illustrate why dynamic cache partitioning, which is the main idea of the proposed algorithm, is required for different workload patterns.

For a heterogeneous mobile storage pairs with a 1.8″ disk and a NAND flash memory, we have evaluated static cache partitioning policies using a cache simulator combined with a multi-device I/O simulator (refer to Sect. 5. In Table 1, the major parameters for the used devices are shown and we set an access time ratio to 35 for the above storage pair for the measurement of the total I/O cost (refer to Sect. 4 for more details) over devices. In experiments, we used both real and synthetic mobile traces for experiments: the PDA trace and trace1 [14]. In the first experiment, we exploited the PDA trace whose working set size is 44 MB while the size of a buffer cache is set as 30 MB. In the second experiment, we used the trace1, whose working set is 23 MB with the buffer cache size fixed at 10 MB. We assume that files are distributed uniformly over different devices of the heterogeneous storage pair.

## 3.1 Effects of Static Cache Partitioning

Figure 1 shows the total I/O costs of several static cache partitioning policies as well as LRU with a single cache partition (Hereafter, LRU indicates the LRU algorithm with a single cache partition unless described otherwise).

In Fig. 1, a different configuration of the cache partition is indicated by a pair (ss, sf) of two values where ss is the cache partition size for a slower device (i.e., a disk) and sf is the cache partition size for a faster device (i.e., a NAND flash). For each cache partition, an LRU policy was used as a cache replacement policy. We notice that as the cache partition size for a disk grows the total I/O cost decreases but has the smallest at a cache partition configuration when ss = 27 MB and sf = 3 MB. When ss is small, LRU outperforms the corresponding static cache partitioning policy, but as it grows (especially after ss goes beyond 18 MB) static partitioning comes to surpass LRU in the aspect of the total I/O cost.

The main reason for LRU's poor performance is that it is based on a single cache partition, so only the temporal locality of each cache block is exploited, making it device-oblivious. That is, LRU is not aware of and has no mechanism of controlling heterogeneity. On the other hand, the static cache partitioning policy has two partitions for a disk and a flash memory and can allocate more blocks to either of the two devices with a selected cache partition configuration although it uses a fixed partition configuration throughout the entire workload. In other words, it is device-aware and can improve the total I/O cost by 63% over LRU with the best cache partition configuration.

The above example shows that using a properly partitioned cache configuration can achieve better I/O performance if we select an appropriate cache partition configuration for a given workload. We notice that device awareness related with workloads has much impact on the performance of a cache partitioning policy and thus should be considered important.

## 3.2 Limits of Static Cache Partitioning

Figure 2 compares best static cache partitioning polices in terms of the total I/O cost in a heterogeneous storage pair of a disk and a flash memory using three different synthetic mobile workloads, SEQ, TEMP, and COMPOUND. SEQ, TEMP, and COMPOUND represent sequential access patterns, access patterns with high temporal locality, and mixed access patterns, respectively. The best static cache partitioning policy means the one which exhibits the smallest total I/O cost for the corresponding workload as the cache partition size varies by a step of 1 MB. In Fig. 2, we notice that there can be a different best cache partitioning configuration for each different workload.

This observation suggests that best static cache partition policies should be varied dynamically according to different workloads. However, I/O access patterns tend to vary even in a given workload, thus making static partitioning less efficient. Therefore, a polished technique rather than just finding a best static cache partition configuration is necessary. Our main goal in this work is, thus, to develop a dynamic partitioning technique which works effectively regardless of given workloads.

## 4. Device-Aware Cache Management Algorithm

### 4.1 Problem Formulation

We assume that a heterogeneous storage system with $n$ different devices $D_i$'s, $i = 1, 2, \ldots, n$. A buffer cache $B$ is divided into $n$ partitions $P_i$'s where $B = \bigcup_{i=1}^{n} P_i$ and $P_i \bigcap P_j = \phi$ for $i \neq j$. We denote $P_i$ to be a partition corresponding to the device $D_i$. $D_i$'s are initially arranged in a descending order of the length of access time. $s(P_i)$ and $m(P_i)$denote the size and cache miss count of the partition $P_i$, respectively. Our assumptions for problem formulation can be summarized as follows.

- We assign a unique cache partition $P_i$ to each device $D_i$. Therefore, we denote the cache partition configuration as $\boldsymbol{P} = \{P_1, P_2, \ldots, P_n\}$. Here, $B = \bigcup_{i=1}^{n} P_i$ and $P_i \bigcap P_j = \phi$ for $i \neq j$.
- The total I/O cost $c(P_i)$ with $P_i$ for a given workload is given by $c(P_i) = R_i \cdot m(P_i)$ where $R_i$ is defined by $R_i =$ (access time of $D_i$)/(access time of $D_n$), $i = 1, 2, \ldots, n$. Since $D_i$'s are arranged in a descending order, $R_i$'s are arranged such that $R_1 \geq R_2 \geq \ldots \geq R_n$.

Then, a generic problem to solve is defined by

Find $P$ that minimizes the sum of the I/O cost per $D_i$

$$c_{tot}(\boldsymbol{P}) = \sum_{i=1}^{n} c(P_i) = \sum_{i=1}^{n} R_i \cdot m(P_i)$$

subject to $B = \bigcup_{i=1}^{n} P_i$ and $P_i \bigcap P_j = \phi$ for $i \neq j$.

Since we focus on a heterogeneous storage pair of a disk and a flash memory in this paper, we limit the problem to solve as follows.

Find $P$ that minimizes the sum of the I/O costs

$$c_{tot}(\boldsymbol{P}) = R_1 \cdot m(P_1) + R_2 \cdot m(P_2)$$

subject to $B{=}P_1 \bigcup P_2$ and $P_1 \bigcap P_2 = \phi$

, where $D_1$ = a disk and $D_2$ = a NAND flash memory.

As we described in Sect. 2, there are two partitioning solutions for this problem: static and dynamic.

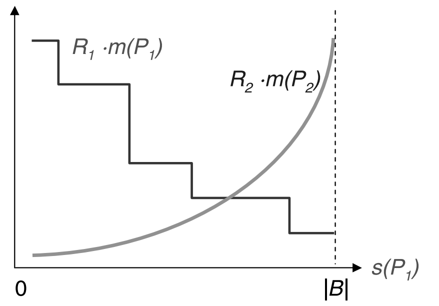


**Fig. 3** Relation between the weighted cache miss counts and cache partition sizes for typical mobile workloads. Since they have mixed I/O access patterns of loop and random accesses, the cache miss counts are likely to be a mix of convex and non-convex functions of the cache partition sizes.

#### 4.1.1 Static Partitioning Solution

Since typical mobile workloads are observed to have mixed I/O access patterns of loop and random accesses [25], the cache miss counts of two devices (i.e., $m(P_1)$ and $m(P_2)$) are likely to be a mix of convex and non-convex functions of the cache partition sizes (i.e., $s(P_1)$ and $s(P_2)$) from the definition of convexity [27]. When only the convexity between individual cache miss counts and cache partition sizes exists, we can achieve an optimal static cache partitioning solution identical to the optimal solution (i.e., the cache partition configuration which causes the optimal total I/O cost).

However, as shown in Fig. 3, since a weighted cache miss count instead of a cache miss count should be used for the total I/O cost across two devices, the overall non-convexity is likely to appear due to more complicated aspects of convex and non-convex functions as the cache partition sizes vary. Therefore, finding a cache partition configuration with the smallest total I/O cost is possible, but this configuration may not be assured to be the optimal solution.

Furthermore, since the static cache partitioning solution uses a fixed cache partition configuration throughout the whole workload, it ignores the necessity of controlling cache partition sizes according to varying workload patterns with time. Thus, the static cache partitioning solution loses many chances of obtaining more optimal cache partition configurations in terms of the total I/O cost.

#### 4.1.2 Dynamic Partitioning Solution

To overcome shortcomings of the static partitioning solution, that is, in order to achieve the optimal total I/O cost flexibly and adaptively for the varying access patterns in a workload as well as for different workloads, we require a dynamic cache partitioning technique. However, it is very hard to obtain the optimal cache partitioning solution during run time. This is because variable access patterns of requests across heterogeneous devices produce dynamically varying functions of cache miss counts to partition sizes and thus exploration of the optimal solution space during run time is almost impossible.

```
1. Monitors $\Delta m(P_1)$, $\Delta m(P_2)$, $\Delta c_{tot}$, and workload patterns periodically.
2. Adapts a weight for increment or decrement of each partition size according to the observed workload patterns.
3. Adjusts the partition sizes ($s(P_1)$ and $s(P_2)$).
4.     If $\Delta c_{tot} < 0$
5.         If $\Delta m(P_1) < 0$ and $\Delta m(P_2) > 0$
6.             Increase $s(P_2)$ with a fine weight.
7.         Else If $\Delta m(P_1) < 0$ and $\Delta m(P_2) < 0$
8.             Increase $s(P_1)$ with a fine weight.
9.         Else
10.             Increase $s(P_1)$ or $s(P_2)$ with a fine weight based on random miss counts of $D_1$ and $D_2$.
11.    Else
12.         Increase $s(P_1)$ or $s(P_2)$ with a fine weight based on random miss counts of $D_1$ and $D_2$.
13.    If $s(P_1) > |B|$, $s(P_1) = |B|$
14.    Else if $s(P_1) < 0$, $s(P_1) = 0$
15.    Else do nothing
16.    $s(P_2) = |B| - s(P_1)$
```

**Fig. 4** Pseudo codes of the proposed dynamic partitioning heuristic.

To deal with finding a reasonable solution dynamically in such a huge search space, several heuristic algorithms have been researched. A well-known method is the simulated annealing algorithm [28]. This algorithm is a probabilistic dynamic approach to find a close-global optimum for a global optimization problem with a large search space. The simulated annealing algorithm may be useful in finding a close-optimal solution, but the process towards the optimal solution depends on probability. Thus, if we would apply it to our dynamic cache partitioning problem, we may obtain a cache partition solution in proportion to the probability in the worst case. Such randomness is not desirable because the blocks stored in the cache partitions will have influence on the total I/O cost as well as next partition control continually. In addition, the simulated anneal approach takes a finer control of parameters near zero temperature, while we cannot know how much the current total I/O cost is close to the smallest value.

Therefore, under the inspiration of the simulated annealing method, we take a simple but effective heuristic dynamic partitioning approach in order to achieve a close-to-optimal total I/O cost rather than the optimal total I/O cost during run time, which depends on the variation of $m(P_1)$ and $m(P_2)$ including workload pattern information. Figure 4 shows pseudo codes of the proposed dynamic partitioning heuristic. This approach tries to keep the descent of $m(P_1)$ (or $m(P_2)$) by adjusting $s(P_1)$(or $s(P_2)$) so that $c_{tot}(\boldsymbol{P}) = R_1 \cdot m(P_1) + R_2 \cdot m(P_2)$ may be minimized. Here, $s(P_1)$ and $s(P_2)$ are the required sizes of $P_1$ and $P_2$ for a close-to-optimal I/O cost. Therefore, actual sizes of $P_1$ and $P_2$ are controlled towards these values when a cache miss occurs across the devices.

In our dynamic partitioning approach, we take two granularities of controlling partition sizes: a coarse-grained adjustment when the total I/O cost increases (i.e., $\Delta c_{tot} \geq 0$) and a fine-grained adjustment when it decreases (i.e., $\Delta c_{tot} < 0$). In the first case our heuristic adjusts the cache partition size based on the random cache miss counts and in the other case it does based on both random cache miss counts and variations of the whole cache miss counts from each cache partition. Similarly to the simulated annealing algorithm, the intuition of our algorithm is to 1) adjust the partition size by a coarse control on the increment in order to have the total I/O cost decrease since it has increased in the first case; 2) control the partition size by a fine control on the increment along with more cases based on more information in order to have the total I/O still decrease or not increase in the other case.

## 4.2 Device-Aware Cache Management Algorithm (DAC)

### 4.2.1 Algorithm Overview

DAC is designed for heterogeneous mobile systems with different storage devices based on the proposed dynamic cache partitioning heuristic. (The current version of DAC targets a pair of a disk and a flash memory. We believe that other different set of devices could be employed if appropriate multi-device I/O simulation would be given) The aim of DAC is to minimize the total I/O cost by 1) adjusting the cache partition using the dynamic partitioning heuristic (we call device-aware cache management) and 2) obtaining performance enhancement by having sequential blocks for a disk and read blocks for a flash memory to be evicted earlier within each partition (we call workload-aware management).

DAC is mainly composed of two levels: 1) adjusting the sizes of partitions for a hard disk and a flash memory; 2) managing each partition according to workload patterns and I/O types together with localities of blocks. The higher level of DAC tracks how the cumulated cache miss count multiplied by the access time during a fixed period of I/O requests has changed for each device and identifies which phase the system is placed in terms of successive total I/O costs. We assume that there are two phases largely: increasing and decreasing phases of the total I/O cost. In the increasing phase of the total I/O cost, DAC controls the cache partition size based on random cache miss counts from each cache partition. This is because the increment of the total I/O cost between successive phases means that the access pattern is likely to have changed and thus random accesses (including both random and loop patterns) are more beneficial than sequential ones. In the decreasing phase of the total I/O cost, DAC mainly controls the cache partition size based on variations of the whole cache miss counts because we infer that the same access pattern of the previous period is likely to continue during this period.

In designing the lower level of DAC, we have two cache management policies: 1) The size of each partition should be adjusted so that sequential accesses may be forwarded to either of the two devices earlier than random ones. 2) Write I/O requests on cache blocks in a flash memory's partition are dispatched to the flash memory later than read ones. Based on these policies, we built an intra-partition

```
VARIABLES AND STRUCTURES
 $x$: referenced block, $|B|$: total cache size
 $W$: partition-adjusting period
 $T1$, $T2$: partitions for a disk and a flash memory
 $cumulated\_ref\_count$: cumulated reference count

MAIN ALGORITHM
 1. Initialize required variables and structures
 2. Determine the access pattern of each block if it is sequential or
random at every reference of blocks
 3. Increase $cumulated\_ref\_count$
 4. If a cache hit occurs in $T1$ (or $T2$)
 5.     Move the block to the MRU in $T1$ (or $T2$)
 6.     Execute H_update
 7. Else
 8.     Execute List_update
 9. If $cumulated\_ref\_count = W$
 10.    Execute the dynamic partitioning algorithm in Fig. 4
```

```
SUBROUTINE List_update
 1. If a miss occurs in $T_j$ ( j = 1 or 2)
 2.     If $|T_j| > T_j\_req$ and $|T_j| \neq 0$
 3.         $T_j.L \leftarrow min\{y.H | y \in T_j\}$
 4.         Evict $y$ which satisfies $y.H = T_j.L$
 5.         Decrease $|T_j|$
 6.         Fetch the block to the cache and move it to the MRU in
$T_j$
 7.         Execute H_update
 8.         Increase $|T_j|$
 9. If $|T_1| + |T_2| \geq |B|$ and $|T_j| > T_j\_req$ (j = 1 or 2)
 10.    $T_j.L \leftarrow min\{y.H | y \in T_j\}$
 11.    Evict $y$ which satisfies $y.H = T_j.L$
 12.    Decrease $|T_j|$
 13.Update random miss count and total miss count in $T_j$
```

```
SUBROUTINE H_update
 1. If a miss occurs in T1
 2.     If block x's pattern is sequential
 3.         $x.H \leftarrow T1.L + C\_SEQ$
 4.     Else
 5.         $x.H \leftarrow T1.L + C\_RAND$
 6. Else
 7.     If $x$'s I/O type is read
 8.         If $x$'s pattern is sequential
 9.             $x.H \leftarrow T2.L + C\_SEQ * C\_READ$
 10.        Else
 11.            $x.H \leftarrow T2.L + C\_RAND * C\_READ$
 12.    Else
 13.        If $x$'s pattern is sequential
 14.            $x.H \leftarrow T2.L + C\_SEQ * C\_WRITE$
 15.        Else
 16.            $x.H \leftarrow T2.L + C\_RAND * C\_WRITE$
```

**Fig. 5** Proposed device-aware cache management algorithm (DAC).

management algorithm, which combines the temporal locality, sequentiality, and I/O types similarly to the Greedy-Dual algorithm within each partition. By controlling each cache partition in a way that the blocks with randomness and high temporal locality stay longer for a disk and the blocks with randomness, high temporal locality, and write I/O types does for a flash, we expect that a large number of sequential requests frequently found in mobile workloads may be managed properly. We also expect that for a flash memory delayed write blocks may cause an effect of reducing write/erase cycles, and thus extend its lifetime and consequently improve the reliability of the overall storage system.

### 4.2.2 Algorithm Details

Detailed DAC's algorithm is shown in Fig. 5. DAC maintains two LRU lists *T1* and *T2*, each of which represents a partition and is assigned to cache blocks for either of heterogeneous devices (presently, *T1* is used for a hard disk and *T2* for a flash memory). *T1_req* (*T2_req*) is the target size of the *T1* (*T2*) partition and $|B|$ is the total cache size. *T1_req* and *T2_req* correspond to $s(P_1)$ and $s(P_2)$, respectively. $W$ represents a partition-adjusting period, which is compared with a cumulated reference count (*cumulated_ref_count*) for re-partitioning.

In the viewpoint of implementation, DAC largely consists of 1) a dynamic cache partitioning algorithm executed at $W$ block references; 2) an intra-partition management algorithm executed at every block reference. At every period of $W$ block references, DAC adjusts the size of each partition dynamically by invoking the dynamic partitioning heuristic algorithm.

At each block access, DAC determines its access pattern. To determine this, we utilized a sequentiality threshold. We set the sequentiality threshold as 4 blocks. This means that if a logical block address (LBA) of a currently accessed block is continued to LBAs of prior accessed blocks by the number of 4 without intermission and formation of any loops, the block is labeled as sequential. Otherwise, it is labeled random. This method is simple but effective and thus many researchers are found to employ it in the literature including [32].

Then, DAC determines whether the access is missed or hit. If a hit occurs in *T1* (or *T2*), the block is moved to the MRU position of *T1* (or *T2*). If there occurs a miss, a subroutine *List_update* is invoked with a flag which indicates in which device the missed block resides. Shortly, the role of *List_update* is to control *T1* and *T2* so that their sizes follow *T1_req* and *T2_req* well, respectively. During such operations, we invoke a subroutine *H_update* to set appropriate values to the cache blocks within each partition according to their attributes.

We subdivided the attribute, which each block can have within the cache, into four: read and sequential, read and random, write and sequential, and write and random. In the algorithm, the attribute is obtained by coupling *C_SEQ*, *C_RAND*, *C_WRITE*, and *C_READ*. Since if a block is identified to be sequential its early eviction is more beneficial in terms of cache hits, *C_SEQ* is set to a smaller value than *C_RAND*. Furthermore, for accesses to blocks residing in a flash memory, we want to delay write I/Os by keeping write blocks longer than read ones and thus *C_WRITE* is set to a larger value than *C_READ*. Therefore, the $H$ values of cache blocks for a disk will be maintained relatively large if they are accessed recently or randomly and they will have more chances to remain in the cache rather than blocks with little locality or sequentiality. In addition to such prioritization on temporal locality and sequentiality, cache blocks for a flash memory will have large $H$ values if each access type is write than when it is read, as shown in *H_update*.

In practice, the intra-partition management algorithm at the lower level of DAC can be designed independently of the dynamic cache partitioning algorithm (i.e., the higher level of DAC). Our workload-aware management algorithm for the intra-partition management algorithm is devised for the purpose of obtaining better performance. To show this, through simulations we compared the results of the original DAC algorithm with those of when LRU is used as the intra-partition management algorithm along with the dynamic partitioning heuristic algorithm.

## 5. Simulation and Results

### 5.1 Simulation Environment

We built a trace-based cache simulator, which simulates LRU, GreedyDual, and DAC, and concatenated it with a multi-device I/O simulator in [14], which can simulate file allocation and I/O accesses over a pair of a hard disk and a NAND flash memory, in order to evaluate the aggregate system I/O performance as well as the total energy consumption. The hard disk model we used is MK4004GAH with a 1.8″ form factor and 4,200 RPM [14] and the flash model is K9K1208U [29].

Our simulator is a trace-driven simulator and processes real file I/O traces from an evaluation board similar to a PDA. This board embeds a 400 Mhz PXA 255 processor, an 10 Mbps ethernet card, a 64 MB SDRAM, and so on. Since we executed real applications, which generate I/O requests, on the board with the features of these devices fixed, all obtained traces can be said to reflect features such as a processor speed, a network bandwidth, and a main memory size in the mobile computing system. Our simulator processes such traces using a buffer cache simulator combined with a multi-device storage system simulator, which includes the performance and energy models for a hard disk and a flash memory (refer to Ref. [14] for more details).

We used two real traces from the evaluation board for evaluation (we call *PDA* and *PMP traces*) and two synthetic traces (we call *trace1* and *trace2*). For generating PDA and PMP traces, we assume that applications running on mobile platforms repeat predefined execution scenarios [7], [14] and the types of applications are limited to *file transfer*, *email*, *file search*, *media play*, and *idling*. The PDA trace simplifies a typical execution pattern of target applications on a PDA and is obtained by repeating file transfer, email, file search, and idling. The PMP trace is obtained by repeating file transfer, media play, and idling to mimic a PMP. These scenarios were performed for 81 and 71 minutes for the PDA and PMP traces, respectively.

To create trace1 and trace2, we built a synthetic trace generator, which can generate three types of traces by controlling sequentiality and temporal locality: SEQ, TEMP, and COMPOUND (In Sect. 3, we have already described these types). Our synthetic trace generator can also control various parameters such as control request rate, read/write ratio, file size, and request size. We ran our trace generator, varying default parameters. The default parameters are set as follows: an average interval time between I/O requests = 70 (ms), a trace time = 80 (min), maximum file size = 5 (MB), a total file size = 350 (MB), and a write ratio = 0.5. The default I/O access pattern is set to COMPOUND (i.e. mixed of sequentiality and temporal locality).

In simulations, trace1 uses the default parameters and trace2 also does except that the average interval time and the maximum file size are set to 20 ms and 1 MB, respectively. The PDA trace, PMP trace, trace1, and trace2 have working set sizes of about 44, 51, 23, and 57 MB and trace file sizes of about 31, 38, 3, and 10 MB, respectively. With these traces, we first evaluated the total I/O costs and flash write counts of DAC, LRU, and GreedyDual. We also compared the total I/O costs of DAC and a static partitioning policy as well as four techniques combining the proposed dynamic partitioning heuristic and different intra-partition management policies. In this paper, since we deal with a performance optimization problem in heterogeneous storage systems, it is very important to determine which performance metric we should use. Using a total I/O cost is one of the most widely-used methods and is an approach which calculates the response time for I/O requests from applications or a kernel onto a storage device to be processed by taking into accounts cache hits or misses at a cache. Many studies in the storage research area have adopted this method as found in [30], [31].

We assumed that the overheads of re-partitioning and handling blocks per partition in DAC are acceptable over LRU. Let the block number of the cache n. LRU needs $O(n)$ steps to check whether a block is hit or not in the LRU list and the block accessed should be moved to the MRU regardless of hit or miss (In our implementation, we used a hash-based look-up for checking hit/miss with $O(1)$ steps). In comparison with LRU, since DAC needs more $O(log(n))$ steps when it searches a minimum H value among cache blocks in the case of eviction we expect that the overhead is sufficiently tolerable due to the limited cache size of $n$.

Finally, we set $W$ to 200 and $R_1$ to 35 and $R_2$ to 1 for DAC, and used the same value of $R_1$ for weighting in GreedyDual. To employ GreedyDual in simulations, we combined different fetch cost per each device and the temporal locality based on LRU, and used a block instead of file data as a caching unit. The method to be used in setting $R_1$ and $R_2$ is similar to those of [30], [31] and employs the fixed ratio between cache and disk access times due to hits and misses at the cache in order to obtain the total response time. Through experiments, we found that the total response time was close-to-optimal when $R_1$ is set to 35. In comparison with our values, Ref. [30] utilized 20 and reference Ref. [31] adopted about 50. $W$ is a period during which the partition sizes should be changed. The $W$ value of 200 was set experimentally so that when there occur 200 block references partition sizes should not be adjusted even if 5 or 6 misses happen in the flash partition for the above $R_1$ and $R_2$ values.

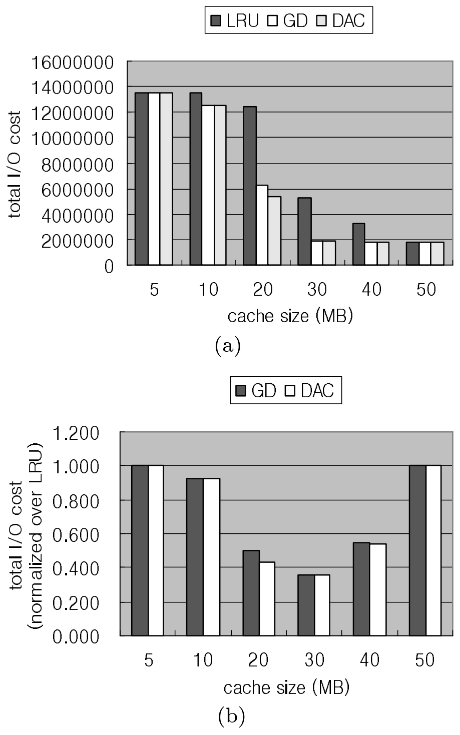


**Fig. 6** For the PDA trace, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.

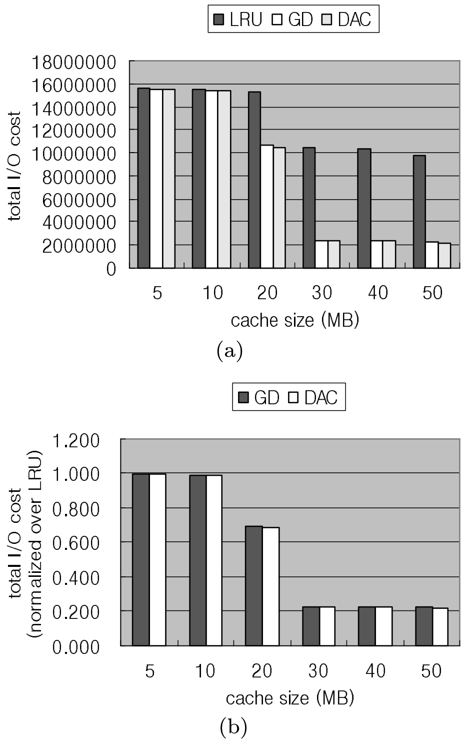


**Fig. 7** For the PMP trace, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.

## 5.2 Simulation Results

### 5.2.1 DAC vs. Existing Caching Algorithms

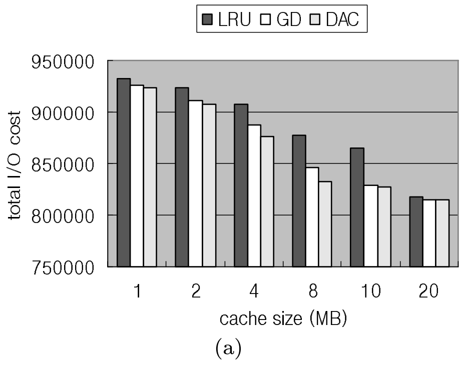
Figure 6 shows the total I/O costs of LRU, GreedyDual (hereafter, we use GD), and DAC, respectively, for the PDA trace as the cache size varies. In the plot (a), DAC shows the smallest total I/O cost among three algorithms with all the cache sizes. DAC improves the overall system I/O performance by up to 64% and 14% over LRU and GD, respectively. This indicates that the proposed device-aware cache management algorithm is more effective in achieving the global optimum of the total I/O cost than other algorithms over the given heterogeneous storage system.

Fig. 6 (b) compares the total I/O costs of GD and DAC through normalization over LRU in order to show how much DAC outperforms GD. Actually, DAC shows the overall performance a little better than GD. With a 20 MB cache size, DAC has the most performance enhancement of about 14% over GD. This is because since the PDA trace has a large number of loop accesses and a considerable number of random ones, and they are accessed uniformly across a disk and a flash, the policy of GD tries to keep more loop blocks accessed onto a disk within the cache longer and evict less loop blocks accessed onto a flash memory earlier and consequently led to a good performance. To determine a best weighting value to a disk blocks (i.e., $R_1$), we tried several values of $R_1$ in evaluating DAC and GD for this trace, but we found that they have little possibilities to obtain improved total I/O costs than when they uses the default $R_1$ value.

Such phenomenon goes further on the PMP trace, as is shown in Fig. 7. Figure 7 shows (a) the total I/O costs of LRU, GD, and DAC and (b) the normalized total I/O costs of GD and DAC for the PMP trace. We notice that DAC still has the best performance among three, achieving I/O cost improvement by up to 77.8% over LRU, but GD shows almost the overall performance enhancement similar to that of DAC. This comes from the fact that the PMP trace also has a lager number of and longer loop accesses than the PDA trace and GD largely tries to cache the blocks from a disk longer.

Figures 8 and 9 show (a) the total I/O costs of LRU, GD, and DAC and (b) the normalized total I/O costs of GD and DAC for the trace1 and trace2, respectively. In Fig. 8, DAC is shown to outperform LRU and GD by narrow margins. This is because the trace1 has been obtained using a synthetic trace generator in a way of randomly being generated, and thus has a lot of random I/O requests and doesn't include loop patterns. Since there are less effects from sequential accesses and more influence from random ones, LRU shows comparable total I/O costs with GD and DAC.

The trace2 has more block accesses with temporal localities than the trace1 although the degree of randomness may be similar between two traces. Therefore, caching disk blocks with localities appropriately along with adjusting the partition size can be beneficial in the aspect of the total I/O cost. From this reason, DAC show the best performance among all the algorithms, as is shown in Fig. 9. DAC im-



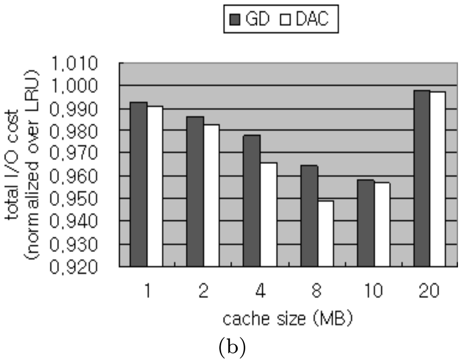


**Fig. 8** For the trace1, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.

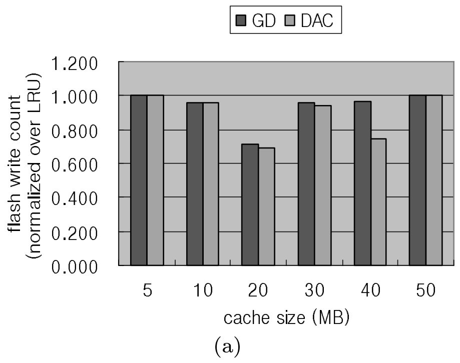
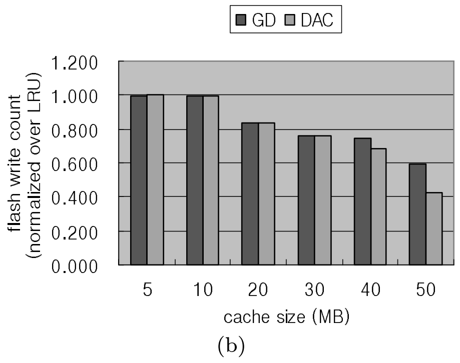



**Fig. 10** (a) For the PDA trace, flash write counts of GD and DAC normalized over LRU (b) For the PMP trace, flash write counts of GD and DAC normalized over LRU.

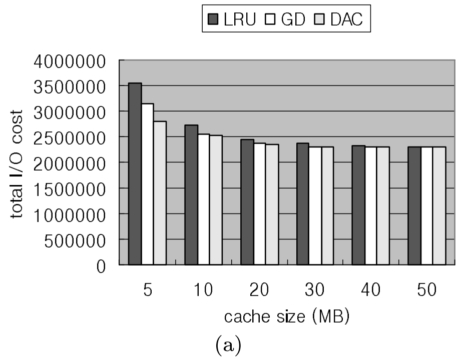
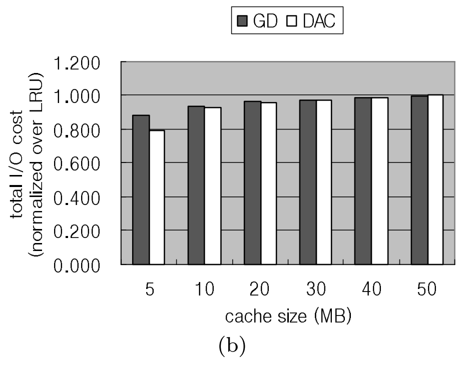



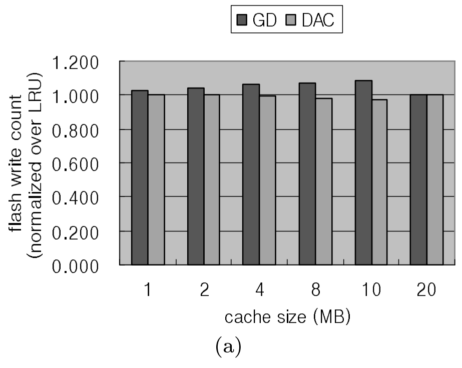
**Fig. 9** For the trace2, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.

proves the total I/O cost by up to 21% and 11% over LRU and GD, respectively.

### 5.2.2 Flash Write Counts and Energy Consumption

Other metrics we evaluated are the number of write accesses onto flash memory and the total energy consumption. The first is important because the maximum number of write/erase cycles is limited in a flash memory and thus reducing the write I/O operations towards a flash memory is likely to have an effect of extending the lifetime of a flash memory. Since we depend on a cache management technique for the purpose of controlling the number of flash writes, the used method may be less aggressive, compared with data migration techniques [14]. The second is also important because a hard disk is a significant energy consumer and thus to increase energy saving by putting the disk into a lower power mode for a long time is a crucial task under a battery-limited environment, although our work does not aim directly at lowering the aggregate energy consumption of both a hard disk and a flash memory. In the viewpoint of price, since we employ a fixed heterogeneous pair of a hard disk and a flash memory device, the total price remains static regardless of the buffer cache algorithm.

Figure 10 shows the number of writes onto the flash memory for the PDA and PMP traces as the cache size varies. In the plot (a), the flash write count of DAC is shown to be smaller than those of LRU and DAC. DAC lowers the number of flash writes by up to 30% and 23% over LRU and



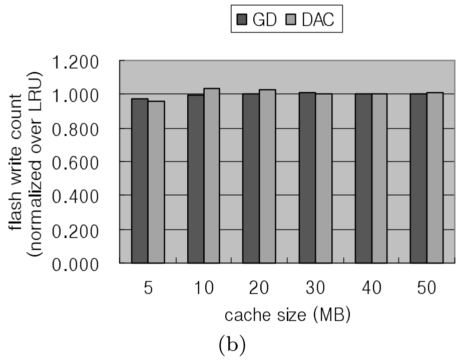
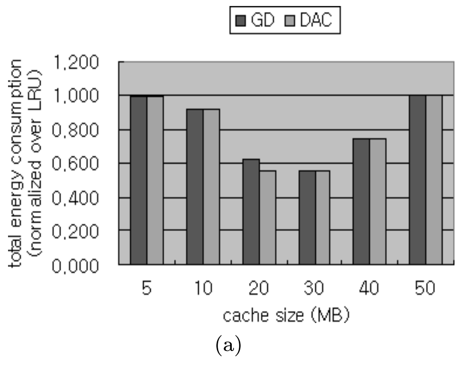


**Fig. 11** (a) For the trace1, flash write counts of GD and DAC normalized over LRU (b) For the trace2, flash write counts of GD and DAC normalized over LRU.



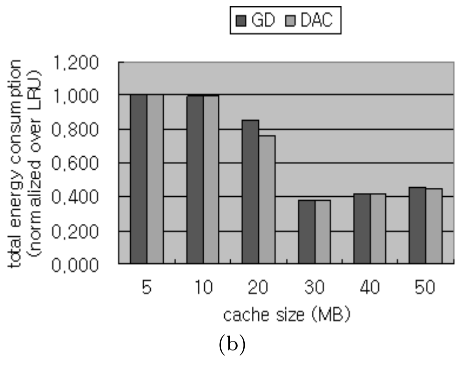


**Fig. 12** (a) For the PDA trace, total energy consumptions of GD and DAC normalized over LRU (b) For the PMP trace, total energy consumptions of GD and DAC normalized over LRU.

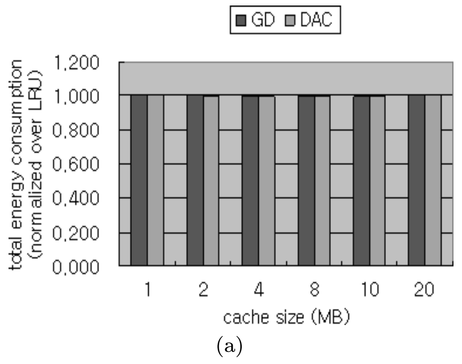
GD, respectively, for the PDA trace. In the plot (b), DAC is shown to reduce the flash write count by up to 57% and 28% over LRU and GD, respectively, for the PMP trace.

To show an effect of reduction of flash writes on extending the flash lifetime, we consider a simple example. We assume that a write ratio is 0.5 (the same value was used in simulations) and an average access arrival rate is 100 accesses/sec. We also assume conservatively that when the amount of data equal to the total capacity is written to the flash memory an erase occurs. If we use LRU as a buffer cache algorithm in a heterogeneous storage system which has a 1 GB flash memory with a 2 KB page and the limited erasure cycles of 10,000 (in case that a flash memory is a multi-level cell (MLC) type), the expected lifetime of a flash memory would take about 6.6 years (10,000*2 GB/100 KB). Since DAC may reduce the flash write accesses by 30% over LRU, the average flash write count will be 35 (= 0.7*0.5*100) and the expected lifetime of a flash memory will be extended to about 9.5 years. If we apply the same calculation in case of using GD, the expected lifetime may amount to about 8.6 years. These values indicate that DAC may cause efficient consumption of write/erasure cycles, a lengthened lifetime of a flash memory, and consequently more reliability into the overall storage system.

Figure 11 shows the number of writes onto the flash memory for the trace1 and trace2 as the cache size varies. Unlike the results of Fig. 10, mitigation of writing onto the flash memory is shown less effective. We guess that random accesses of these two traces and another randomness used to assign write I/O type to each access may disturb DAC's attempt to evict write blocks later. Actually, we noticed that the plot (a) shows DAC still outperforms LRU slightly, but the flash write counts of DAC in the plot (b) were shown to be rather larger than those of LRU. For the trace2, this seems that a larger number of read or write blocks within the flash partition with high temporal localities evict write blocks often and thus the flash write count cannot be made small.

Figure 12 shows the total energy consumed by both the hard disk and the flash memory for the PDA and PMP traces as the cache size varies. In Fig. 12, we can notice that GD and DAC consume much less energy that LRU, saving energy by up to 44% and 58% for the PDA and PMP traces, respectively. In the plot (a), DAC is shown to lower the total energy consumption by about 11% over GD at the cache size of 20 MB for the PDA trace. In the plot (b), DAC is also shown to reduce the total energy consumption by up to 11% over GD for the PMP trace. Good energy savings of DAC can be attributed to considerably reduced I/O accesses onto the hard disk. Due to the reduced I/O accesses, the total spin-down times of DAC, during which the hard disk stays in the standby mode mode (refer to Table 1.) are revealed to be more than 3 times longer than those of LRU for the PDA and PMP traces. Therefore, DAC has more chances to save much energy than LRU.

Figure 13 shows the total energy consumption for the trace1 and trace2 as the cache size varies. Unlike the results of Fig. 12, energy saving is shown less effective. Especially,



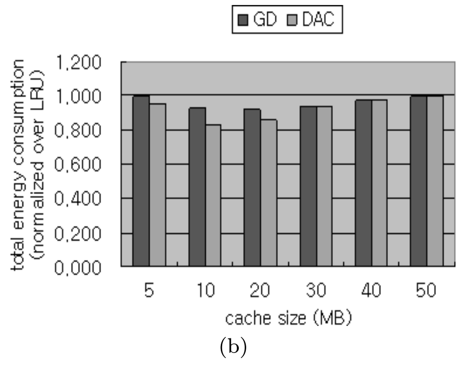
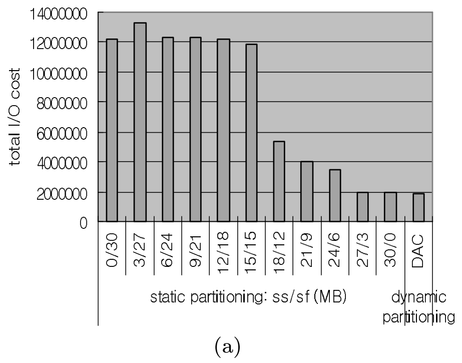


**Fig. 13** (a) For the trace1, total energy consumptions of GD and DAC normalized over LRU (b) For the trace2, total energy consumptions of GD and DAC normalized over LRU.



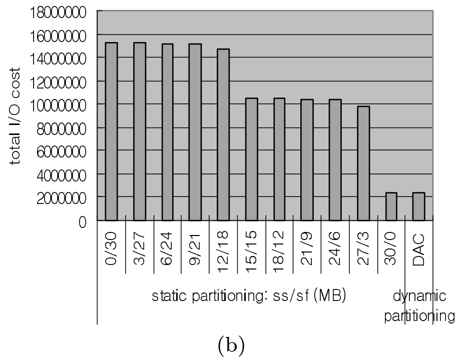
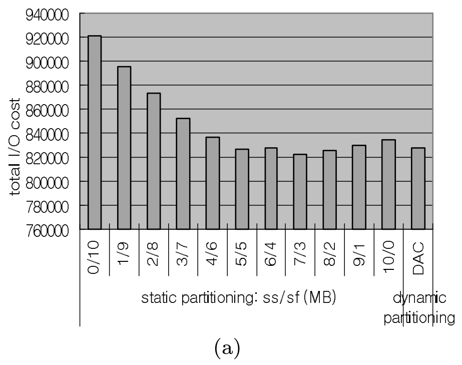


**Fig. 14** Total I/O costs of DAC and static partitioning, which uses a fixed static partitioning policy and the intra-partition management of DAC (a) for the PDA trace, (b) for the PMP trace.



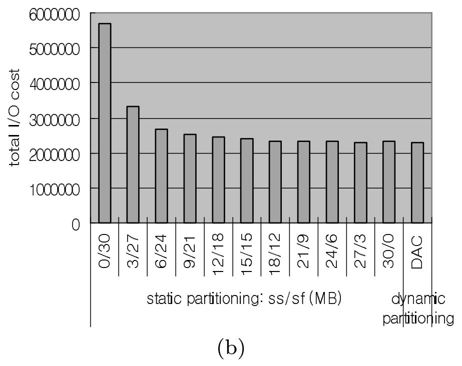


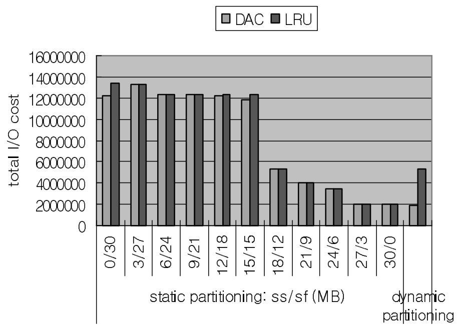
**Fig. 15** Total I/O costs of DAC and static partitioning, which uses a fixed static partitioning policy and the intra-partition management of DAC (a) for the trace1, (b) for the trace2.

the plot (a) shows that DAC has almost the same energy consumption as GD or LRU. This is because, since the trace1 have a small working set and its I/O accesses have temporal localities, the total standby time was shown similar regardless of the buffer cache algorithm. The total energy consumption of DAC in the plot (b) was shown to be smaller than LRU by up to 17%.

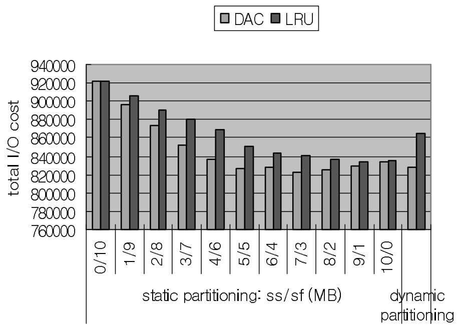
### 5.2.3 DAC vs. Static Cache Partitioning

We compared the total I/O costs of DAC and static cache partitioning techniques as the cache partition size varies for the PDA, PMP, trace1, and trace2, as shown in Figs. 14 and 15, respectively. Static cache partitioning employed in this experiment uses a fixed static partitioning policy and the intra-partition management of DAC. That is, we just replaced the dynamic cache partitioning heuristic of DAC with a fixed static partitioning policy in this experiment.

We observed that DAC has smaller total I/O costs than static partitioning techniques for all the traces except trace1. In Fig. 14 (a), we can notice that there occurred the smallest total I/O cost at the partition configuration at (27 MB, 3 MB) for a disk and a flash memory with a 30 MB total cache among the graphs of static partitioning techniques. A similar result could be found in Fig. 14 (b) for the PMP trace though deviation at partition configurations with the smallest static total I/O cost occurred, where a partition for a disk rather than a flash memory is large. But, in (a) and (b), the total I/O costs with these best static partition con-



(a)



(b)

**Fig. 16** Total I/O costs of four techniques: the static partitioning policy + the intra-partition management algorithm of DAC, DAC, and the dynamic cache partitioning of DAC + LRU (a) for the PDA trace, (b) for the trace1.

figurations were found to be larger than those of DAC for each trace. Therefore, we believe that DAC can adapt different workload patterns dynamically and achieves a more close-to-optimal performance than static cache partitioning techniques.

In Fig. 15, we also found that static cache partitioning techniques with the smallest total I/O costs for each trace: in (a) at the partition configuration of (7 MB, 3 MB) with a 10 MB total cache for the trace1; in (b) at the partition configuration of (27 MB, 3 MB) with a 30 MB total cache for the trace2. Although DAC has a little larger (less than 1%) total I/O cost than the static partitioning technique with the minimal total I/O cost (i.e., partitioning the cache as 7 MB for a disk and 3 MB for a flash) in (a) for the trace1, DAC still can be said to find close-to-optimal performance by dynamically adjusting the required partition size per each device for better performance based on varying access patterns. For the trace2, DAC shows better system I/O performance than static partitioning with the minimal total I/O at (27 MB, 3 MB) of a partition configuration for a disk and a flash memory.

### 5.2.4 Comparisons of Four Techniques

Figure 16 compares the total I/O costs of four techniques: the static partitioning policy + the intra-partition management algorithm of DAC, the static partitioning policy + LRU, DAC, and the dynamic cache partitioning heuristic of DAC + LRU for (a) the PDA trace and (b) trace1, respectively. We can notice that DAC, which uses dynamic cache partitioning and workload-aware intra-partition management, shows better total I/O cost than static cache partitioning + LRU or the dynamic cache partitioning heuristic of DAC + LRU. (We already mentioned the comparison between the results of DAC and the static partitioning policy + the intra-partition management algorithm of DAC previously.) In Fig. 16 (a), since the PDA trace has a large number of loop accesses and a smaller number of random ones and access patterns are varying continuously, the dynamic cache partitioning algorithm seems to have a great contribution rather than the intra-partition management algorithm. Compared with this, since accesses consist of mostly random ones and there are less workload pattern fluctuations in Fig. 16 (b), the intra-partition management algorithm seems to have a high impact on the total I/O cost relatively though the dynamic cache partitioning algorithm still have significant influence.

### 5.2.5 Implementation Issue

An implementation of DAC remains one of our major future researches and we thus consider an implementation issue shortly in this subsection. The following requirements should be met for the purpose of the implementation of the propose buffer cache algorithm. First, a hard disk and a flash memory device which utilize the same file system and host interface should be used. For example, a heterogeneous storage pair of a small form-factor hard disk and a solid-state disk (SSD) using IDE interfaces is eligible. Second, since the proposed buffer cache algorithm should be implemented at the level of the operating system, we need the accessibility of source codes of the target operating system. Unlike Linux, source codes of MS Windows are not open and thus actual implementation is almost impossible for Windows. Third, we need to analyze the buffer cache management mechanism and file system of the target operating system. Since processing data on file accesses is required we need to implement the proposed algorithm at the level of the file system. Then, using proper data structures like linked lists replacing the existing buffer cache algorithm with the proposed one will be possible. For example, files are managed in pages at the buffer cache and thus we should determine whether the page size is 4 KB or 8 KB and how the corresponding page will be managed within the buffer cache at cache hit/miss.

Let us take an example for Linux 2.6. In Linux 2.6, a buffer cache uses two LRU lists (active and inactive) and checked whether a hit or a miss occurs at the buffer cache in the function called *generic_file_read*. When read requests are invoked from applications a system call in the kernel is invoked and then *generic_file_read* is called. Linux 2.6 defines and utilizes a data structure named *address_space* to manage paged within the buffer cache. This structure contains lists called *clean page*, *dirty page*, and *locked page* as members. Linux provides read, write, and setdirty functions for these members and uses a radix tree as the kernel should search all pages within the buffer cache fast when it checks whether a page is a hit or a miss at the buffer cache. In order to implement the proposed algorithm in Linux 2.6, we need to change the *address_space* data structure. In detail, we should add an L value to the *inode* data structure and *List_update* and *H_update* functions should be implemented. We need to insert a routine which determines each block's access pattern. Finally, a routine adjusting the size of each partition with a period of *W* is required.

## 6. Related Work

There has been a lot of research in mobile storage systems by combining hard disks with flash memory in terms of performance enhancement and/or energy saving. [5]–[7], [11], [13] have all proposed using flash memory as a non-volatile cache, keeping blocks which are likely to be accessed in the near future, and thus allowing a hard disk to spin down for longer time. [6], [13] focused on usage of a flash memory as a write buffer cache along with enhanced spin-down techniques, while [7] has recently studied a technique of partitioning the flash memory into a cache, a prefetch buffer, and a write buffer to save energy. [11] mainly considered reducing power consumption of the main memory using a flash memory as a second-level buffer cache.

The Hybrid Hard Disk Drive technology co-developed by Samsung and Microsoft uses a flash memory as an onboard non-volatile cache in addition to a hard disk, which aims at performance boosting, low power, and high reliability on mobile computers [8]–[10]. In the meantime, Intel has developed the Robson technology [12], which uses also a flash memory as a non-volatile cache to increase system responsiveness, make multi-tasking faster, and extend battery life time in combination with the new features of Windows Vista like ReadyDrive and ReadyBoost. The Hybrid Hard Disk Drive technology focuses on the benefits of proximity between a HDD and a flash memory and uses an SATA interface, while the Robson technology uses a PCI-like interface.

Unlike the above researches, the proposed cache management technique targets mobile systems with a heterogeneous secondary storage pair consisting of a hard disk and a flash memory instead of using a flash memory as a non-volatile cache. It tries to enhance the overall system I/O performance by applying an effective buffer cache management algorithm on the mobile workloads, which is able to consider both device-awareness and workload-awareness.

In disk-based storage systems, [23] studied storage-aware cache management algorithms using different cost on heterogeneous disks. This work maintained one partition per each disk and adjusted partition sizes based on the time spent per each disk over a fixed period of time (they call this wait time). But, their algorithms cannot detect the case of there being a number of sequential accesses because they control the blocks within each partition using LRU or CLOCK algorithms, which may be problematic. This is because if a considerable number of sequential accesses are requested to a disk its wait time can be lengthened by filling the corresponding partition with less valuable blocks.

Recently, [25] have proposed a cost-aware cache replacement algorithm, which targets mobile systems with a pair of a hard disk and a flash memory as secondary storage. This work partitions the cache per device by exploiting a performance index combined with cache miss counts and access time per device and further manages each partition in order to achieve the fast sequential performance feature of a hard disk.

While [23] and [25] focused on the work imbalance problem, which may occur different I/O costs of heterogeneous storage devices, by enlarging the partition for a slower device, our work concentrates on optimizing the overall I/O cost itself. This is necessary because some mobile workloads can still exhibit critical work imbalance due to skewed device access counts. In comparison with [25], we not only have our algorithm manage cache partitions based on the sequentiality of I/O requests but also mitigate write/erase cycles of a flash memory in consideration of I/O types

## 7. Conclusions

We have proposed a novel buffer cache management algorithm which considers both I/O cost per device and workload patterns in mobile computing systems with a heterogeneous storage pair of a hard disk and a NAND flash memory. In order to minimize the total I/O cost under varying workload patterns, the proposed algorithm employs a dynamic cache partitioning technique over different devices and manages each partition according to request patterns and I/O types along with the temporal locality.

Trace-based simulations show that the proposed algorithm improves the total I/O cost by up to 77.8% and 14% and flash write count by up to 57% and 28% over LRU and GreedyDual, respectively, on typical mobile traces, when a 1.8″ hard disk and a NAND flash memory are employed. It also achieves as good performance as the best static cache partitioning policy almost always.

We have several future work plans. First, we plan to study device-aware cache management on various heterogeneous storage devices including MEMS-based storage and a hard disk. Second, we plan to data migration techniques along with our cache algorithm to obtain high performance as well as low energy consumption. Third, we consider studying a dynamic cache partitioning algorithm in consideration of prefetching. Finally, implementation and verification in the real file systems are other important challenges.

## Acknowledgments

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No.R0A-2007-000-20116-0), the Brain Korea 21 Project in 2008, and the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) Support program supervised by the IITA (Institute of Information Technology Advancement)(IITA-2008-C1090-0801-0020). The ICT at Seoul National University provides research facilities for this study. Young-Jin Kim was also supported by the Sun Moon University Research Grant of 2008.

**References**

[1] Y. Shin, "Non-volatile memory technologies for beyond 2010," Proc. 2005 Symposium on VLSI Circuits Digest of Technical Papers, 2005.

[2] L.R. Carley, G.R. Ganger, and D.F. Nagle, "MEMS-based integrated-circuit mass-storage systems," Commun. ACM, vol.43, no.11, pp.72–80, Nov. 2000.

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

[4] Samsung Electronics, MLC NAND Flash memory (K9HBG08-U1M) and SLC NAND Flash memory (K9WBG08U1M), Data Sheets, 2006.

[5] B. Marsh, F. Douglis, and P. Krishnan, "Flash memory file caching for mobile computers," Proc. 27th Hawaii International Conference on System Sciences, pp.451–460, Hawaii, USA, Jan. 1994.

[6] T. Bisson and S. Brandt, "Reducing energy consumption with a non-volatile storage cache," Proc. International Workshop on Software Support for Portable Storage (IWSSPS), held in conjunction with the IEEE Real-Time and Embedded Systems and Applications Symposium (RTAS 2005), San Francisco, California, March, 2005.

[7] F. Chen, S. Jiang, and X. Zhang, "SmartSaver: Turning flash drive into a disk energy saver for mobile computers," Proc. 11th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'06), Tegernsee, Germany, Oct. 2006.

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

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

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

[11] T. Kgil and T. Mudge, "FlashCache: A NAND flash memory file cache for low power web servers," Proc. 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06), Seoul, Korea, Oct. 2006.

[12] M. Trainor, "Overcoming disk drive access bottlenecks with Intel Robson Technology," Technology Intel Magazine, Dec. 2006. http://www.intel.com/technology/magazine/computing/robson-1206.htm

[13] 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.

[14] 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.

[15] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance NAND flash-based storage system," J. Syst. Archit., vol.53, no.9, pp.644–658, Sept. 2007.

[16] M. Uysal, A. Merchant, and G.A. Alvarez, "Using MEMS-based storage in disk arrays," Proc. 2nd. USENIX Conference on File and Storage Technologies (FAST), March-April 2003.

[17] F. Wang, B. Hong, S.A. Brandt, and D.D.E. Long, "Using MEMS-based storage to boost disk performance," Proc. 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005), Monterey, CA, USA, April 2005.

[18] Samsung Unveils its Third Fusion Semiconductor - Flex-OneNAND$^{TM}$. http://www.samsung.com/us/business/semiconductor/newsView.do?news_id=810

[19] New Toshiba mobileLBA-NAND memory chips for mobile phones support both SLC and MLC memory areas. http://www.toshiba.com/taec/news/press_releases/2007/memy_07_482.jsp

[20] ActiveShopper. http://www.activeshopper.com

[21] inSpectrum Tech. Inc. NAND Flash price. http://www.inspectrumtech.com/DP/NANDFlashContractPrice.aspx

[22] 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.

[23] B. Forney, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Storage-aware caching: Revisiting caching for heterogeneous storage systems," Proc. 1st. USENIX Conference on File and Storage Technologies (FAST), Jan. 2002.

[24] E. Pinheiro, R. Bianchini, E.V. Carrera, and T. Heath, "Load balancing and unbalancing for power and performance in cluster-based systems," Proc. International Workshop on Compilers and Operating Systems for Low Power, Sept. 2001.

[25] 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.

[26] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," Proc. USENIX Symposium on Internet Technology and Systems, Dec. 1997.

[27] R.G. Bartle, The Elements of Real Analysis, John Wiley & Sons, 1976.

[28] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by simulated annealing," Science, vol.220, no.4598, pp.671–680, 1983.

[29] H.G. Lee and N. Chang, "Low-energy heterogeneous non-volatile memory systems for mobile systems," Journal of Low Power Electronics, vol.1, no.1, pp.52–62, April, 2005.

[30] G. Yadgar, M. Factor, and A. Schuster, "Karma: Know-it-all replacement for a multilevel cache," Proc. 5th USENIX Conference on File and Storage Technologies (FAST), pp.169–184, 2007.

[31] T.M. Wong and J. Wilkes, "My cache or yours? Making storage more exclusive," Proc. USENIX Annual Technical Conference (ATC), 2002.

[32] 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, Oct. 2000.

**Young-Jin Kim** received the B.E. degree and the M.E. degree in electrical engineering and the Ph.D. degree in computer science & 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 a faculty member in the Department of Computer Science and Engineering, Sun Moon University, Asan, Korea. His research interests include low-power embedded systems, low-power storage systems, 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.