Energy-Aware Flash Memory Management in Virtual Memory System

Han-Lin Li, Chia-Lin Yang, *Member, IEEE*, and Hung-Wei Tseng

***Abstract*—The traditional virtual memory system is designed for decades assuming a magnetic disk as the secondary storage. Recently, flash memory becomes a popular storage alternative for many portable devices with the continuing improvements on its capacity, reliability and much lower power consumption than mechanical hard drives. The characteristics of flash memory are quite different from a magnetic disk. Therefore, in this paper, we revisit virtual memory system design considering limitations imposed by flash memory. In particular, we focus on the energy efficient aspect since power is the first-order design consideration for embedded systems. Due to the write-once feature of flash memory, frequent writes incur frequent garbage collection thereby introducing significant energy overhead. Therefore, in this paper, we propose three methods to reduce writes to flash memory. The HotCache scheme adds an SRAM cache to buffer frequent writes. The subpaging technique partitions a page into subunits, and only dirty subpages are written to flash memory. The duplication-aware garbage collection method exploits data redundancy between the main memory and flash memory to reduce writes incurred by garbage collection. We also identify one type of data locality that is inherent in accesses to flash memory in the virtual memory system, intrapage locality. Intrapage locality needs to be carefully maintained for data allocation in flash memory. Destroying intrapage locality causes noticeable increases in energy consumption. Experimental results show that the average energy reduction of combined subpaging, HotCache, and duplication-aware garbage collection techniques is 42.2%.**

***Index Terms*—Embedded systems, energy-efficient, NAND flash memory, virtual memory.**

I. Introduction

THE MODERN operation system often adopts the virtual memory approach to overcome the limitation on physical memory size and allow the physical memory shared among multiple tasks as well. The traditional virtual memory system is designed for decades assuming a magnetic disk as the secondary storage. Recently, flash memory has become a popular storage alternative for many portable devices with the continuing improvements on its capacity, reliability, and much lower power consumption than mechanical hard drives. The characteristics of flash memory are quite different from a magnetic disk. Therefore, in this paper, we revisit virtual memory system design considering limitations imposed by flash memory. In particular, we focus on the energy efficient aspect since power is the first-order design consideration for embedded systems.

TABLE I
NAND Flash Characteristics

operation	latency	energy consumption
page read	47.2 μs	679 nJ
page write	533 μs	7.66 μJ
block erase	3 ms	43.2 μJ

There are two main types of flash memory. One is NOR flash memory, and the other is NAND. NAND flash memory is commonly used for data storage due to its lower cost and higher density compared with NOR flash. NAND flash memory is composed of blocks, and each block contains a set of pages. The typical block and page size are 16 kB and 512 B, respectively. There are three types of operations in flash memory: read, write and erase. The energy consumption of these operations is shown in Table I. A page is the basic unit for read/write operations. Due to the write-once feature, a page cannot be overwritten. Therefore, flash memory performs out-place updates. That is, data is written to a free page, and the old page is invalidated. Those with invalid data are called dead pages. After a certain number of writes, free space on flash memory would be low. When the free space is lower than a threshold value, flash memory must reclaim dead pages through erase operations. Such reclaiming process is called garbage collection. Erasing is done in a unit of one block. Since a block consists of multiple pages, live pages of the victim block must be copied to free space before the block is erased. This is considered as garbage collection overhead. Frequent garbage collection does not only incur significant energy overhead, it also shortens the lifetime of flash memory since flash memory has limited number of erase operations. To achieve efficient garbage collection, ideally, we want to find a victim block with only dead pages. In this way, we could release most free pages with least copying overhead through garbage collection. Therefore, one key principle to achieve efficient garbage collection is to allocate data accessed close in time (i.e., locality) to the same flash block [1], [21]. Since these data will be invalidated together after a period of time, we are more likely to find a victim block with only very few live pages for garbage collection.

From the above discussion, we know that reducing writes to flash memory is critical for energy optimization of flash memory. Eliminating writes not only reduces energy consumed

Manuscript received February 2, 2007; revised May 30, 2007 and July 11, 2007; accepted September 15, 2007. This work was supported in part by the National Science Council of Taiwan under Grant NSC 95-2221-E-002-098-MY3, Grant NSC 96-2752-E-002-008-PAE, and Grant NSC 96-2221-E-002-250- and in part by the Excellent Research Projects of National Taiwan University under Project 95R0062-AE00-07. This paper was presented in part at the 2006 International Symposium of Low Power and Electronics Design .

H.-L. Li and C.-L. Yang are with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan, R.O.C. (e-mail: yangc@csie.ntu.edu.tw).

H.-W. Tseng is with the Department of Computer Science and Engineering at University of California at San Diego, La Jolla, CA 92093 USA (e-mail: h1tseng@cs.ucsd.edu).

Digital Object Identifier 10.1109/TVLSI.2008.2000517

for handling writes. More importantly, it results in less frequent garbage collection which could be a significant part of flash memory energy once flash memory has been used for a period of time. Reducing writes can be done in two ways: one is to filter out unnecessary write requests from the OS kernel; the other is to reduce copying overhead from garbage collection. In this paper, we exploit these two flash properties for energy optimization of the secondary storage in the virtual memory system. We make the following contributions.

1) We identify one type of data locality that is inherent in accesses to flash memory in the virtual memory system. In the virtual memory system, a page fault results in series of flash writes. We refer to this locality as intrapage locality. For efficient garbage collection, we would like to allocate flash pages of the same memory page to the same flash block. Destroying intrapage locality (i.e., flash pages of the same memory page are scattered in two different blocks) could potentially result in significantly higher energy consumption of flash memory. One interesting example is that the threshold value triggering garbage collection could affect how well intrapage locality is preserved during garbage collection and for subsequent writes. We observe that two garbage collection threshold values differing only by one could actually result in more than 40% energy differences.
2) We study the effect of subpaging to filter out unnecessary writes to flash memory. In the traditional virtual memory system, the full victim page is written back to the disk when a page miss occurs. While this is OK for a disk, it is not energy-efficient for flash memory. A 4 K virtual memory page results in eight writes to flash memory assuming a 512 B flash memory page. We observed that a victim page often contains unmodified data. Therefore, writing a full page results in unnecessary writes to flash memory. Although subpaging reduces writes to flash memory, intrapage locality no longer exists. If the energy benefit of reducing writes through subpaging cannot compensate the adverse effect of losing intrapage locality, subpaging could possibly increase energy consumption of flash memory. The experimental results show that subpaging reduces flash energy by 15.8% on average assuming a main memory pages contain eight flash pages.
3) We propose to add an SRAM to cache frequent writes to flash memory. This cache is called HotCache in this paper. To increase cache utilization, we investigate two approaches for HotCache management. In the first approach, every write is cached and the replacement policy is based on both the access time and frequency factors (TF policy). The second approach is to identify frequently written data and only those data are stored in the cache. One problem particular to flash cache management is that higher cache hit rate does not necessarily result in more energy savings. In addition to cache hit rate, preserving intrapage locality when writing back data from HotCache to flash memory is critical. The experimental results show that the TF policy with page locality gathering can achieve 16.3% energy reduction with a 1 M cache.
4) We exploit data redundancy between the main memory and flash memory to reduce the copying overhead of garbage collection. When a main memory page is swapped in, this page exists in both the main memory and flash storage. This page will be written back to flash memory if it is dirty when it gets swapped out next time. The old copy in flash memory becomes dead pages. Therefore, we could reduce garbage collection overhead by not copying these flash pages of a victim block. The experimental results show duplication-aware garbage collection reduces flash energy by 24.1% on average.

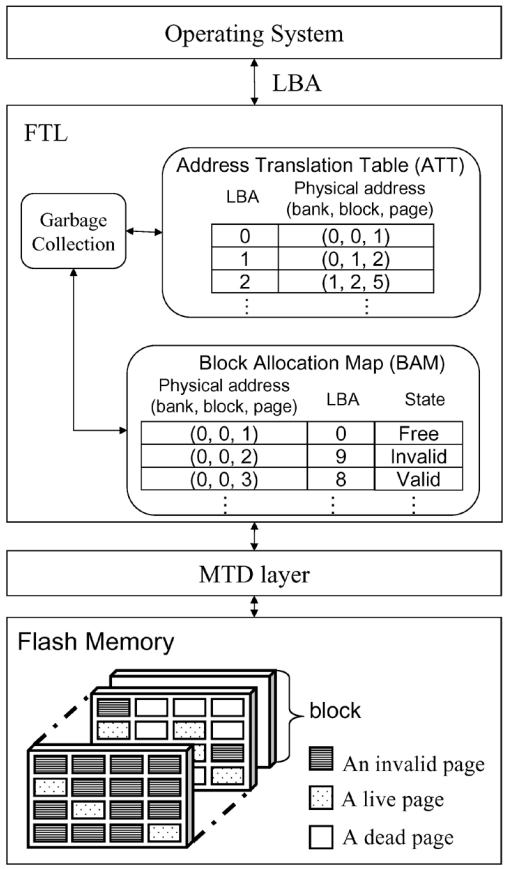


Fig. 1. The flash memory storage system architecture.

The rest of the paper is organized as follows. Section II introduces background knowledge of flash memory. Section III then describes important characteristics of the virtual memory system using flash memory as a swap device. The details of the proposed energy efficient flash memory management is presented in Section IV. Experimental methodology and results are described in Section V and Section VI, respectively. Section VII discusses related work. Finally, Section VIII concludes this paper.

II. Background

Fig. 1 shows the architecture of the flash memory storage system. The flash memory storage system consists of the flash translation layer (FTL), memory technology device (MTD) driver, and the flash memory chips. Data in flash memory is identified by operating system using LBAs (logical block addresses) to emulate a block device. FTL handles address translation between LBAs and physical address on flash

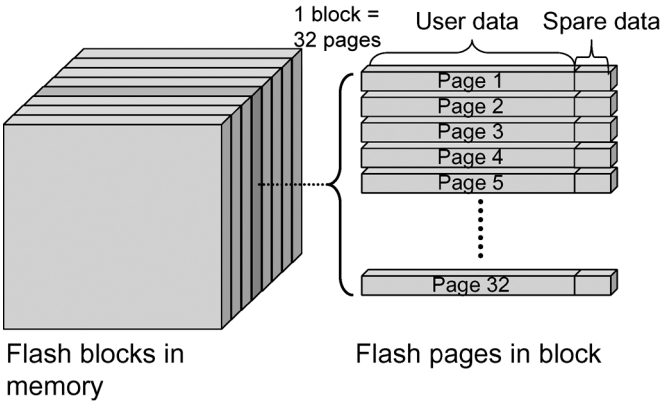


Fig. 2. NAND flash architecture.

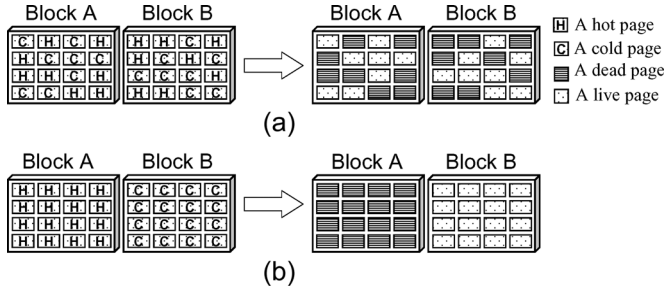


Fig. 3. Data allocation (a) without and (b) with hot–cold separation.

memory. Besides, FTL is also responsible for garbage collection and data allocation on flash memory. The MTD layer provides handling routines for read, write and erase operations between flash memory and FTL. In this section, we detail the flash memory architecture, garbage collection policy, and data allocation method for efficient garbage collection.

A. Flash Memory Architecture

Fig. 2 shows a typical architecture of a NAND flash memory. A NAND flash memory is organized in blocks, and each block contains a fixed number of pages. A block is the smallest unit for erase operations, while reads and writes are processed in terms of pages. The block and the page size of a typical NAND flash memory are 16 kB and 512 B, respectively. There is a 16-byte spare area appended to every page. Bookkeeping information, such as a page's logical block address and erase counts, are stored in spare areas. Each page and its spare area can be read/written independently, and they are wiped together on erase. Each flash block has a limited number of erase operations. With current technology, a block of a typical NAND flash memory could be erased for 1 million times. A block is considered as being worn out if its erase cycle count exceeds the limitation. Once a block is worn out, it could suffer from frequent write errors.

Due to the write-once feature, flash memory performs out-place updates. A written page can not be rewritten unless it is erased. When the data on a page are updated, the new data is written to free space and the old copies of the data are invalidated. A page is referred to as a live page if it contains valid data, and a dead page if it contains invalidated data. A dead page becomes a free page through an erase operation. After a certain number of write operations, the free space on flash memory will be low, garbage collection will be triggered to reclaim dead pages by erasing a block, which is called the victim block. The live pages of the victim block must be copied to free pages before a block being erased.

Because of out-place update and garbage collection, the physical location of an LBA changes from time to time. FTL provides transparent address translation between LBAs and physical address by using a RAM-resident translation table. The translation table is indexed by LBA, where each triple (bank_num,block_num,page_num) indicates its corresponding bank number bank_num, block number block_num, and page number, page_num. The table can be rebuilt at system boot time by scanning the block allocation map (BAM) which is stored in flash memory. BAM is an array indexed by physical address, and keeps the page state (e.g., free, valid or dead).

B. Garbage Collection (GC)

Garbage collection is triggered when the free space of flash memory is equal to a predefined threshold value. Since valid pages of a victim block must be copied before being erased, one of the goals of garbage collection is to minimize the copying overhead. A well-known garbage collection policy is the greedy policy which always recycle the block with the largest number of dead pages. The greedy policy is proven to be efficient when data are uniformly accessed. If the workload has some frequently updated data, which is known as hot data, the garbage collection policy should avoid copying hot data since such data would usually become invalid soon. However, the greedy policy is not aware of data update frequency therefore cannot avoid copying hot data. To solve this problem, the cost-benefit policy is proposed [7]. The cost-benefit policy reclaims the block with largest cost-benefit which is calculated as

$$\text{age} \times \frac{1-u}{2u}$$

where age stands for the time past since last modification of the block (i.e., the last page write or invalidation), and u for the live pages in the block. The term $2u$ represents the cost for copying (u for read valid pages and u for write back these pages), and $1-u$ represents the free spaces reclaimed. The cost-benefit policy avoids recycling a block containing recently invalidated data because the policy considers that more data on the block will be invalidated soon. Since the cost-benefit policy is more efficient than greedy, we adopt the cost-benefit policy as our default garbage collection policy in this paper.

C. Data Allocation: Locality Gathering

One commonly used approach to reduce copying overhead of garbage collection is to allocate frequently written data (hot data) in the same block. This is also called locality gathering. Consider the example shown in Fig. 3. In Fig. 3(a), hot data are scattered in two blocks, while in Fig. 3(b), hot data are clustered in one block. We assume that after a period of time, all hot data are accessed, and the cold data remain valid. For the data allocation where hot data are clustered [Fig. 3(b)], block A contains

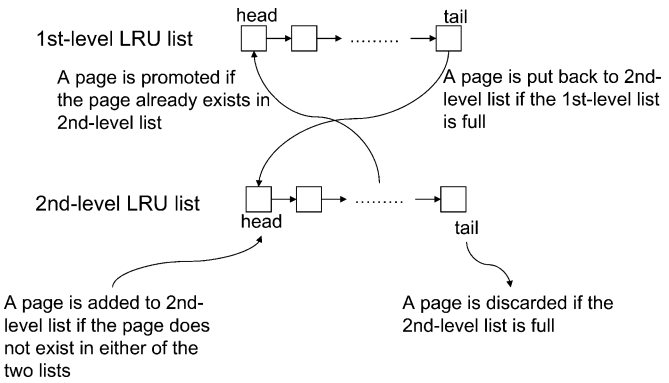


Fig. 4. Hot–cold separation mechanism: two-level LRU.

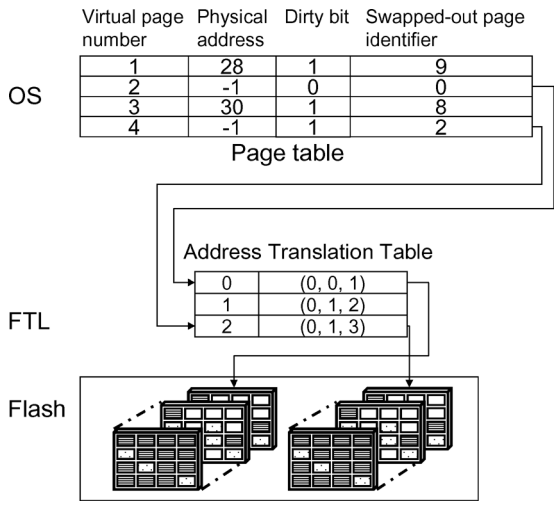


Fig. 5. Flash memory as a swap device.

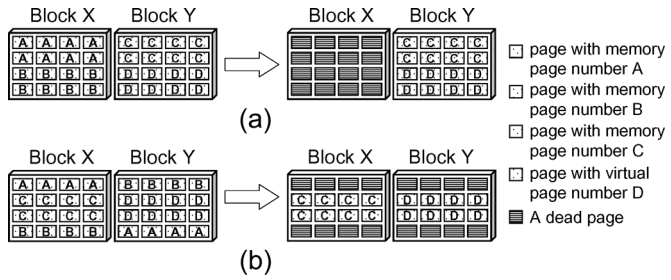


Fig. 6. Data allocation with/without intrapage locality.

only dead pages. Therefore, it incurs zero copying overhead if the block A is selected as the victim. As for the data allocation shown in Fig. 3(a), dead pages are distributed in two blocks. Therefore, selecting either block as the victim incurs copying overheads. Reducing garbage collection overhead has significant impact on energy since it results in less frequent garbage collection. Therefore, data allocation with locality gathering is critical for designing an energy efficient flash storage.

Several studies have proposed methods to perform locality gathering. The eNVy system [21] proposes to allocate hot data towards the lower numbered blocks while cold data in the opposite direction. In eNVy, data are always written to the tail of a block, therefore, data near the end are considered as hot data. During garbage collection, pages at the head of the victim block are copied to higher numbered block, while pages near the end are copied to the opposite direction.

The other approach for locality gathering is the hot–cold separation mechanism proposed by Chang *et al.* [1]. They use two-level LRU lists for hot/cold data identification as shown in Fig. 4. An LBA is first inserted into the second-level LRU, and is promoted to the first-level list if the LBA is written again before being evicted from the second-level list. If the first-level list is full, the last element of first-level list is put back to the second-level list. The LBAs recorded in the first-level list are considered as hot data. Two pointers are maintained to record the blocks that are currently used to store hot and cold data.

III. Flash Memory as Swap Device

Fig. 5 shows the virtual memory system using flash memory as the secondary storage. In the virtual memory system, on a page fault, if the faulting page exists in the swap device, the kernel obtains its LBAs from the page table. The page slot in the swap area is reused as long as it is not overwritten. So a clean victim page is not written back to the swap device if its page slot is not overwritten. When a page fault occurs, if the victim page is dirty or it does not exist in the swap device, a series of writes are issued to flash memory. The faulting page is then swapped in through a series of read requests. For a 4 k main memory page, and 512 B flash page, a page fault incurs eight reads/writes to flash memory. That is, flash pages of a main memory page are always accessed back to back. This is referred to as intrapage locality in this paper. From the discussion in Section II-C, we understand the importance of allocating data that are accessed close in time to the same flash block. With the same reasoning, preserving intrapage locality is important for efficient garbage collection. Consider two data allocation methods shown in Fig. 6. Assume a main memory page contains eight flash pages. In Fig. 6, two flash blocks contain four main memory pages A, B, C, and D. In Fig. 6(a), flash pages in one main memory page are allocated in one block, while in Fig. 6(b), they are scattered in two blocks. Assume after a period of time, memory page A and B are swapped out. Therefore, for the data allocation in Fig. 6(a), block X contains only dead flash pages, while for the data allocation in Fig. 6(b), both block X and Y contain dead and live pages. Therefore, destroying intrapage locality could cause adverse effect on garbage collection efficiency thereby increasing overall energy consumption.

One example that affects how well intrapage locality is preserved is the threshold value GC_t that triggers garbage collection. Garbage collection occurs when the number of free pages is equal to GC_t. Fig. 7 shows the overall flash energy consumption normalized to the case $\mathrm{GC}_t = 256$ with GC_t varying from 255 to 288 for mozilla, a web browsing application. This set of experiment assumes that a flash block contains 32 flash pages, and a virtual page contains eight flash pages. We can observe that for threshold value 256, 264, 272, 280, and beyond, we get lowest normalized energy. To analyze this behavior, we divide different GC_t values in three categories. In the discussion below, we assume that a main memory page contains m flash pages, a flash block contains n flash pages, $n \bmod m = 0$, and intrapage locality is preserved before the first garbage collection.

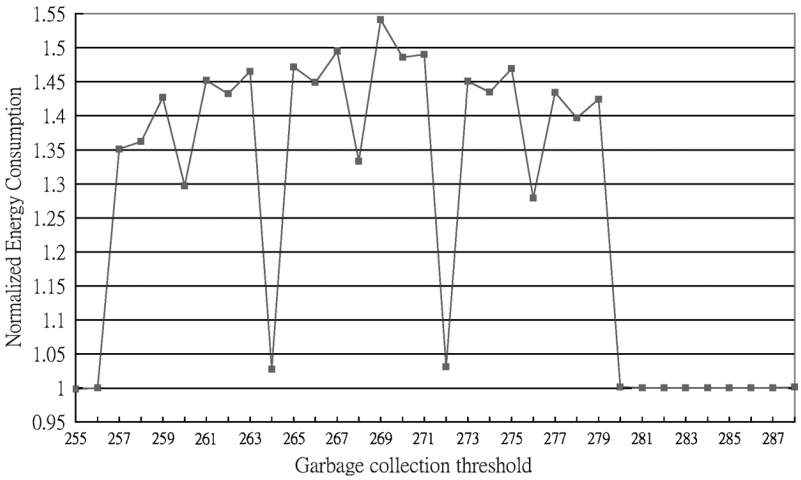


Fig. 7. Overall flash energy consumption under different GC thresholds.

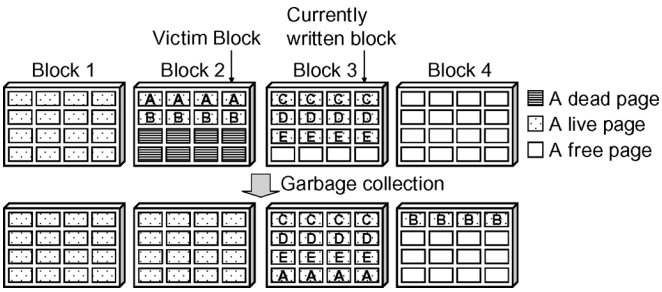


Fig. 8. Case1: $n = 16$, $m = 4$, and $\mathrm{GC}_t = 20$.

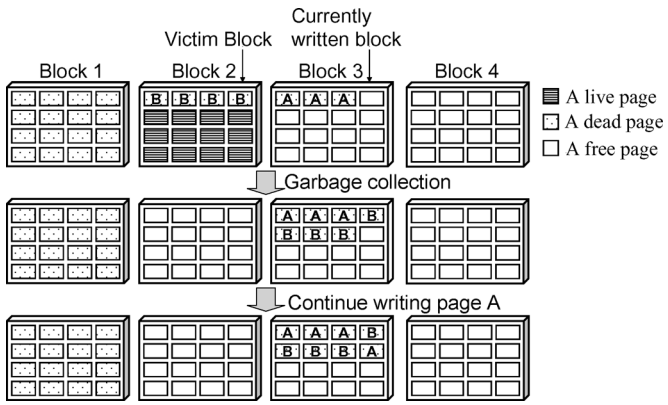


Fig. 9. Case2: $n = 16$, $m = 4$, and $\mathrm{GC}_t = 25$.

- $\mathrm{GC}_t \bmod m = 0$
  Fig. 8 illustrates the case where $n = 16$, $m = 4$, and $\mathrm{GC}_t = 20$. Since $\mathrm{GC}_t \bmod m = 0$, when garbage collection is triggered, the number of free pages of the currently written block (block 3 in this example) must be a multiple of m. Since data are invalidated in the granularity of m pages, the number of live pages of the victim block must be also a multiple of m. Therefore, we could guarantee that intrapage locality is not destroyed by garbage collection. In the example shown in Fig. 8, flash pages of main memory page A and B are allocated to the same block, respectively. Moreover, the number of live pages in block 4 (currently written block) is a multiple of m after garbage collection so the intrapage locality of subsequent writes will be preserved.
- $\mathrm{GC}_t \bmod n \geqq n - m$
  In this case, the number of free pages of the currently written block is always more than $n-m$ pages, and garbage collection is triggered in the middle of writing back a main memory page. Fig. 9 illustrates this scenario where block 3 is the currently written block, and garbage collection occurs while the main memory page A is written back to flash memory. Since the maximal number of live pages of the victim block should not be more than $n - m$ according to the cost-benefit policy described in Section II-B, these live pages are guaranteed to be allocated to block 3. After garbage collection, FTL continues writing the remaining flash pages of the main memory page A to block 3. So intrapage locality is preserved, and the live pages of block 3 after garbage collection is still a multiple of m. Therefore, the intrapage locality of subsequent writes will be preserved.
- $\mathrm{GC}_t \bmod m \neq 0$ and $\mathrm{GC}_t \bmod n < n - m$
  For threshold values that do not meet the above two conditions, if the currently written block has more free pages than live pages of the victim block, intrapage locality will be preserved during garbage collection. Otherwise, one live page of the victim block will be forced to split into two different blocks. Fig. 10 illustrates this scenario. Assume $\mathrm{GC}_t = 23$, and garbage collection occurs while the main memory page E is written back to flash memory. Since the free pages of block 3 is less than the live pages of the victim block, pages B are allocated in block 3 and 4. After garbage collection, FTL continues writing the remaining flash pages of main memory page E to block 4. So the intrapage locality of page E and B are destroyed. Furthermore, the intrapage locality of subsequent writes will also be destroyed if either page E or B gets invalidated later and the corresponding block is selected as the victim for garbage collection. In the example shown in Fig. 10, let us assume

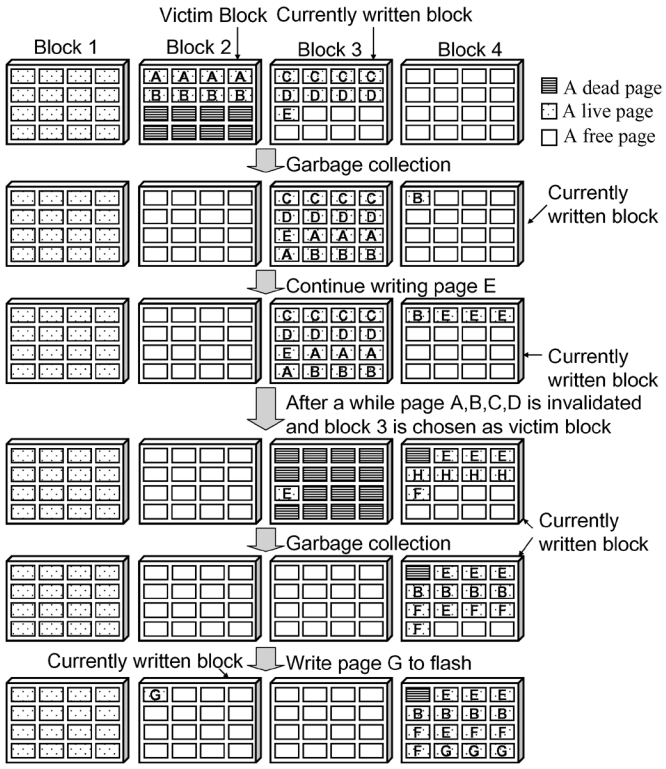


Fig. 10. Case3: $n = 16$, $m = 4$, and $\mathrm{GC}_t = 23$.

pages A, B, C, D are invalidated. Garbage collection is triggered while writing back page F, and block 3 is chosen as the victim. So live page E is copied to block 4, and writing main memory page F continues after garbage collection. We can see that after writing main memory page F completes, the number of live pages of the currently written block (block 4) is no longer a multiple of m. So for the subsequent write, page G in this example, its intrapage locality cannot be preserved.

IV. Energy-Efficient Flash Memory Management

In this section, we describe three techniques to reduce the energy consumption of flash memory in the virtual memory environment. To reduce unnecessary writes to flash memory, we divide a main memory page into a set of subpages, and only dirty subpages are written into the flash memory when a page fault occurs. An SRAM is used to cache frequently written data, which is called HotCache. The third technique is duplication-aware garbage collection.

A. Subpaging

In the traditional virtual memory system, a full victim page is written back to the disk when a page miss occurs. A typical flash page size is 512 B or 2 kB, while a main memory page size could be 4 kB, 2 MB, or 4 MB. With a 4 kB virtual memory page and 512 B flash page, each page fault incurs eight writes to flash memory. For the applications tested in this paper, we find that the victim page often contains a significant amount of unmodified data. Table II shows the ratio of dirty blocks in a victim page assuming a 512 B block, and 4 K virtual memory page. We can see that only very few applications, e.g., gqview, have high dirty ratio. For mozilla, the dirty ratio is below 50%. Therefore, writing a full victim page to flash memory is not energy efficient.

TABLE II
Dirty Ratio of Workloads[1]

application	dirty ratio	application	dirty ratio
kword	89.73%	kword+juk	69.41%
mozilla	48.49%	mozilla+juk	40.90%
kspread	88.61%	kspread+juk	72.40%
openoffice	66.59%	openoffice+juk	59.31%
gqview	98.86%	gqview+juk	97.62%

The subpaging technique divides a virtual memory page into a set of subpages. The subpaging technique was previously proposed to reduce the transferring size and latency of remote memory in a networked system [5]. To tailor the subpaging technique for flash memory, we divide a page in the granularity of flash page size. Each subpage is associated with a dirty bit. On a page fault, only dirty subpages are written into flash memory. Park *et al.* [10] proposed a new replacement policy, clean first least recently used (CFLRU), to reduce writes to flash memory by keeping dirty pages in memory as long as possible. Although this method reduces the energy consumption of flash memory effectively, it could incur more page faults. In contrast, the subpaging technique reduces writes to flash memory without increasing page faults.

B. HotCache

To reduce writes to flash memory, we propose to keep frequent writes to an SRAM, which is referred to as HotCache in this paper. There exist products that integrate SRAM and flash memory in one package [15]. HotCache is organized as a fully associative cache with the HotCache block size equal to the page size of flash memory.

The HotCache management policy affects the performance of the HotCache. eNVy [21] proposed to use an SRAM as a write buffer. That is, every write request is cached and the first in first out (FIFO) policy is adopted for replacement. In this paper, we investigate three new policies for the HotCache management. Below we detail these three policies.

Time-Frequency (TF): In the Time-Frequency policy, every write request is cached in HotCache. The replacement is based on the following weight function:

$$\text{timestamp} \times \text{write\_counts}.$$

The HotCache block with the smallest weight is selected as the victim when a replacement occurs. This policy considers both the time and frequency factors. The advantage of this TF policy over the traditional LRU is that it prevents a hot page from being replaced by a recently accessed cold page.[2]

Time-Frequency-Locality (TFL): With the TF policy described above, the flash pages of a virtual memory page is not guaranteed to be allocated in the same block since they may

[1]The dirty ratio is defined as the number of dirty 512 B block in a dirty memory page/the number of 512B blocks in a main memory page.

[2]Hot (cold) pages are those frequently (rarely) accessed.

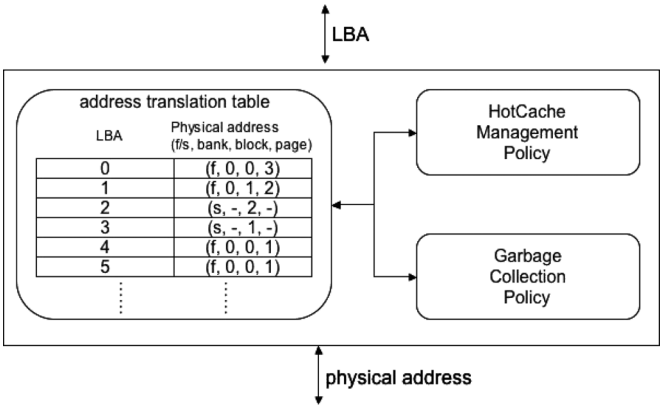


Fig. 11. Modified FTL for HotCache.

not be replaced out of the HotCache in sequence. Therefore, to avoid destroying intrapage locality, we enhance the TF policy by forcing all the pages of the same virtual memory page to be replaced in sequence. The virtual page number of the victim block is recorded and a counter is used to keep track of how many HotCache blocks in the same main memory pages have been replaced. A HotCache block with the smallest timestamp $\times$ write_counts and has the same recorded virtual page number is chosen as the victim HotCache block. Once the counter reaches zero, a HotCache block with smallest timestamp $\times$ write_counts is selected as the victim. Its main memory page number is then recorded and the counter is reset. Note that since the virtual memory page number of a HotCache block can be obtained directly from the cache tag,[3] we do not need to record the page number of each HotCache block. The enhanced TF policy is called TF-locality (TFL).

Two-Level LRU (2L): Different from the TF and TFL policies, the two-level LRU policy observes a page for a period of time to determine whether this page should be allocated in HotCache. Similar to hot/cold data separation policy proposed by Chang *et al.* [1]. we use the first-level LRU list records the pages considered as hot data, and the second-level list records the pages considered as the candidates to be hot data. The difference is that we allocate hot data, which pages are recorded in first level list, to HotCache, and cold data are written to flash memory.

The length of the first-level LRU is the number of HotCache blocks. That is, every HotCache block has a corresponding entry in the first level LRU list. Note that the 2L policy does not destroy the intrapage locality since it considers only the time factor for replacement. Therefore, flash pages in the same virtual page are guaranteed to be replaced back to back.

Modifications to Support HotCache: To support the HotCache architecture, modifications across each layer of flash storage system are also required. FTL needs to handle the HotCache management as shown in Fig. 11. The write handling routine is modified to support the HotCache as follows: if the requested LBA exists in the HotCache, the write is performed directly in the HotCache. Otherwise, if the HotCache management policy determines that the LBA should be allocated in the HotCache, the block in flash memory is invalidated, data are written into the HotCache, and the replaced HotCache block is written back to the flash. Algorithm 1 summarizes the modified write handling routine.

[3]virtual page number = cache tag $\times$ flash page size $\div$ main memory page size.

Algorithm 1: Handle Write Request

1: **if** LBA exists in HotCache **then**
2: update the LBA on HotCache
3: **else**
4: **if** HotCache_Policy(LBA) = HotCache **then**
5: write back victim(HotCache)
6: invalidate the original copy
7: write LBA in HotCache
8: **else**
9: write to flash memory
10: **end if**
11: **end if**

Because the physical address has been changed from a triple to a 4 tuple, the MTD layer must be enhanced to handle the address. As MTD receives an physical address belongs to HotCache, MTD can translate the read/write requests into SRAM access commands.

C. Duplication-Aware Garbage Collection (DA-GC)

DA-GC is to exploit data redundancy between the main memory and flash memory to eliminate unnecessary live page copying during garbage collection. When a main memory page is swapped in, this page exists in both the main memory and flash storage. This page will be written back to flash memory if it is dirty when it gets swapped out next time. The old copy of flash memory becomes dead pages. Therefore, we could reduce live page copying in garbage collection by not writing these flash pages of a victim block to free space. Therefore, in the proposed duplication-aware garbage collection scheme, when garbage collection occurs, FTL does not move live pages that are found in the main memory to free space. To ensure the correctness of the swap system adopting DA-GC, those flash pages that are omitted during garbage collection are guaranteed to be written back to the flash memory when they are swapped out of the main memory. DA-GC reduces writes to those flash pages that are dirty at the time of garbage collection or will become dirty later. For those pages remaining clean to the time they get swapped out of the main memory, DA-GC delays those writes that would have occurred during garbage collection to the swap-out time.

To realize this idea, as shown in Fig. 12, we add three fields in the BAM of FTL: PID (process id), virtual page number (VPN) and the in-memory flag. When a dirty (or new) memory page is swapped out, the swap system sends a write request to FTL along with this page's LBA, process identifier, and virtual page number. FTL searches the BAM with the requested LBA. The process identifier and virtual page number is recorded in the matched entry, and the in-memory flag is cleared to indicate that the corresponding flash page does not exist in the

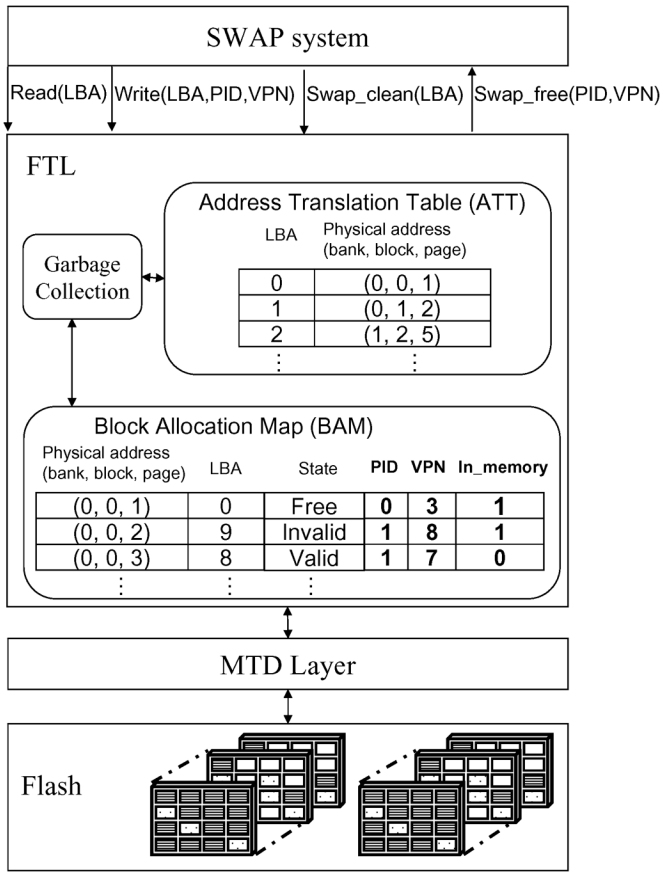


Fig. 12. DA-GC.

main memory. When a page is swapped into the main memory, FTL sees a read request, and sets the corresponding in-memory flag. Note that in the conventional swap system, FTL is not aware of the swap-out events of clean pages. Therefore, to support DA-GC, we add a swap_clean call to notify FTL the swap_out event of a clean page. The other issue that we need to address to support DA-GC is to guarantee the write back of those flash pages that are omitted during garbage collection. To achieve this, FTL notifies the kernel of the associated PID and virtual page number of omitted flash pages with a swap_free call which sets the dirty bit of the matched PTE (page table entry). In the case of shared memory, swap_free may fail to find the matched PTE. Assume memory page X is shared between process A and B. If process A is selected by the kernel to swap out page X, process A's identifier and corresponding virtual page number are recorded in the BAM. Assume memory page X is later swapped in by process B. Since the BAM is indexed with LBAs, the in-memory flag can be set correctly. When the flash pages of memory page X are omitted during garbage collection, FTL calls swap_free with process A's identifier. If process A has not read memory page X, the corresponding PTE is actually invalid. In this case, swap_free should return a failure to disable DA-GC. The other case that swap_free needs to return a failure is when a process detaches shared memory. The detach operation invalidates corresponding PTEs of the shared memory. Therefore, swap_free is not able to find the matched PTEs if the associated shared page has been detached.

TABLE III
WORKLOADS USED IN OUR EXPERIMENTS AND THEIR CHARACTERISTICS

Application	Memory footprint	write/total inst.	Scenario
kword	39.2 MB	15.7%	edits several lines and reform the columns in a document
mozilla	42.1 MB	12.9%	browses websites including Yahoo, Google, Gmail, Amazon, and etc.
kspread	28.4 MB	11.7%	calculates sum, average, min, max of numerical data and sorts them.
openoffice	74.2 MB	14.3%	plays about twenty slides.
gqview	52.9 MB	32.5%	views some images
juk	19.9 MB	13.8%	plays a list of mp3 files summing up to 1 hour

TABLE IV
SIMULATION PARAMETERS

device	operation	energy consumption
SRAM (512 KB) [17]	access energy	1.82 nJ
SRAM (1 MB) [17]	access energy	3.02 nJ
NAND (page size = 512B) [14]	page read	679 nJ
	page write	7.66 μJ
	block erase	43.2 μJ
NAND (page size = 2KB) [16]	page read	2.36 μJ
	page write	14.5 μJ
	block erase	54.0 μJ

V. EXPERIMENTAL METHODOLOGY

We adopt a trace-driven simulation in this paper. Our simulator contains a main memory paging system and flash storage with an SRAM. We use Valgrind [9] on an x86-linux machine to collect memory traces. The applications we tested in this study are listed in Table III: kword, a word processor; mozilla, a web browser; kspread, a spreadsheet application; openoffice, a popular office suite similar to Microsoft office; gqview, a image viewer. We also create multiprogramming workloads by running juk, an MP3 jukebox program, with applications listed in Table III. The OS adopts the round-robin scheduling policy.

We assume the physical memory allocated to user programs is 16 MB, and the main memory page size is 4 kB. The virtual memory is managed using the LRU policy. For flash memory, we assume a 16 kB block with 512 B page size or a 128 kB block with 2 kB page size. We adopt the cost-benefit policy [7] as our garbage collection policy. We assume the initial utilization of flash memory is 97%. The SRAM sizes of the HotCache considered in this paper are 512 kB and 1 MB.

The access energy of SRAM and flash memory assumed in our experiments are listed in Table IV. The SRAM access energy is obtained using the CACTI [17] assuming 0.13 μm technology. The CACTI is an integrated cache access time, power and area model, and has been widely used in studies on cache architecture [4], [18]. The energy consumption of flash memory are based on the data sheet of Samsung K9F1208R0B [14] (page size $=$ 512 B) and K9K2G08X0A [16] (page size $=$ 2 kB) NAND flash.

VI. SIMULATION RESULTS

A. Subpaging

Fig. 13 shows the energy consumption of the subpaging technique normalized to the baseline storage system (without

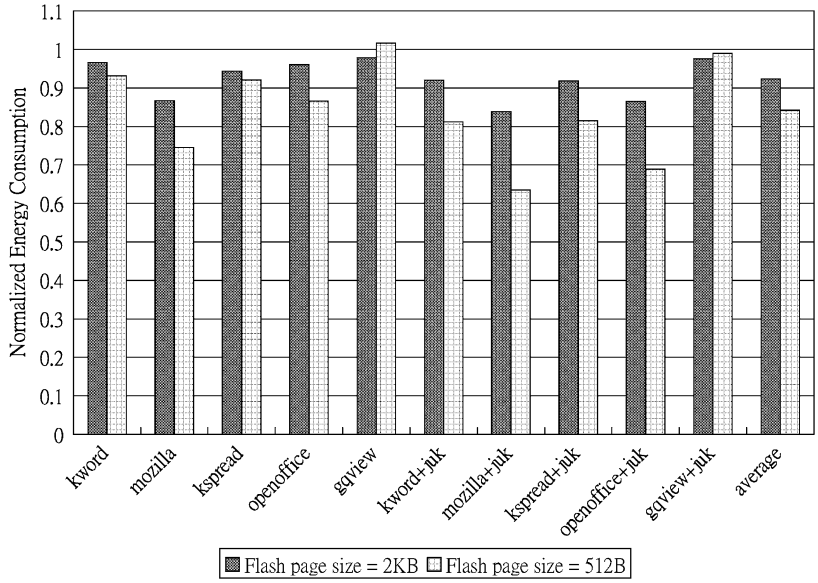


Fig. 13. Relative energy consumption of subpaging technique in 512 B and 2 kB flash page size.

TABLE V
WRITE HIT RATES OF HOTCACHE MECHANISMS (HOTCACHE SIZE: 512 kB/1 MB)

Application	512KB					1MB				
	FIFO	LRU	2L	TF	TFL	FIFO	LRU	2L	TF	TFL
kword	0.33%	0.35%	2.24%	1.78%	3.86%	1.24%	1.37%	2.24%	3.73%	6.14%
mozilla	0.05%	0.06%	8.29%	7.47%	8.22%	1.11%	1.10%	15.01%	27.64%	13.72%
kspread	0.09%	0.09%	1.09%	2.26%	5.43%	5.64%	6.60%	2.48%	33.45%	9.36%
openoffice	0.40%	0.62%	5.77%	6.18%	3.65%	3.80%	4.50%	10.43%	11.40%	6.19%
gqview	0.00%	0.00%	0.05%	0.14%	0.61%	0.03%	0.12%	0.05%	0.56%	1.69%
kword+juk	0.68%	1.23%	8.87%	10.27%	7.12%	6.20%	7.52%	16.94%	17.56%	11.79%
mozilla+juk	0.05%	0.06%	6.07%	6.20%	5.67%	0.55%	0.79%	11.28%	12.39%	10.64%
kspread+juk	1.92%	1.81%	7.71%	8.69%	5.99%	9.79%	10.69%	14.35%	15.84%	8.62%
openoffice+juk	0.57%	0.67%	7.00%	7.51%	3.94%	4.95%	6.34%	13.37%	14.35%	5.76%
gqview+juk	0.00%	0.01%	0.11%	0.22%	0.80%	0.05%	0.10%	0.11%	0.61%	2.00%
average	0.41%	0.49%	4.72%	5.07%	4.53%	3.34%	3.91%	8.62%	13.75%	7.59%
average normalized energy	0.945	0.953	0.919	0.964	0.909	0.877	0.880	0.853	0.858	0.837

subpaging) assuming both the 512 B and 2 kB flash page size. We also show write reduction rates in Table VI. We can see that with a 512 B flash page, the subpaging could reduce up to 59.1% writes and 36.5% of flash energy (mozilla+juk). Since there are only few writes in gqview, the subpaging technique only reduces 1.1% of writes. Note that gqview shows slight increases in energy consumption after applying subpaging. Although subpaging reduces writes to flash memory, intrapage locality no longer exists. If the energy benefit of reducing writes through subpaging cannot compensate the adverse effect of losing intrapage locality, as in the case of gqview, subpaging could increase energy consumption. With a 2 kB flash page, the subpaging is less effective since a main memory page only contains two flash pages. The effects of subpaging depends on the dirty ratio which is defined in Section IV-A. The lower the dirty ratio is, the higher energy reduction we expect to see by adopting subpaging. In single-programming workloads, mozilla and openoffice show most significant energy reduction via subpaging since the dirty ratio of these two applications are much lower than others. Multiprogramming workloads show lower dirty ratio than single-programming ones since there are more contention for the memory resource. With a 512 B flash page, the subpaging achieves about 22.2% energy saving on the average for multiprogramming workloads, and 10.4% for single-programming workloads.

TABLE VI
REDUCED WRITES BY SUBPAGING

	page size			page size	
application	512B	2KB	application	512B	2KB
kword	10.3%	3.7%	kword+juk	30.6%	11.5%
mozilla	51.5%	18.2%	mozilla+juk	59.1%	22.7%
kspread	11.4%	3.9%	kspread+juk	27.6%	11.1%
openoffice	33.4%	15.5%	openoffice+juk	40.7%	19.3%
gqview	1.1%	0.4%	gqview+juk	2.4%	0.9%
overall average				26.81%	10.72%

B. HotCache

In this section, we evaluate the HotCache hit rates and energy efficiency of various caching policies discussed in Section IV-B. We also show two commonly used policies: FIFO and LRU. Table V list the write hit rates and energy reductions of a 512 kB and 1 MB HotCache, respectively.

From Table V, we can see that the TF policy has the highest HotCache hit rate, 5.07% for a 512 kB cache and 13.75% for a 1 MB cache. The FIFO policy used in eNVy [21] has A much lower hit rate compared with the TF policy, 0.41% for a 512 B

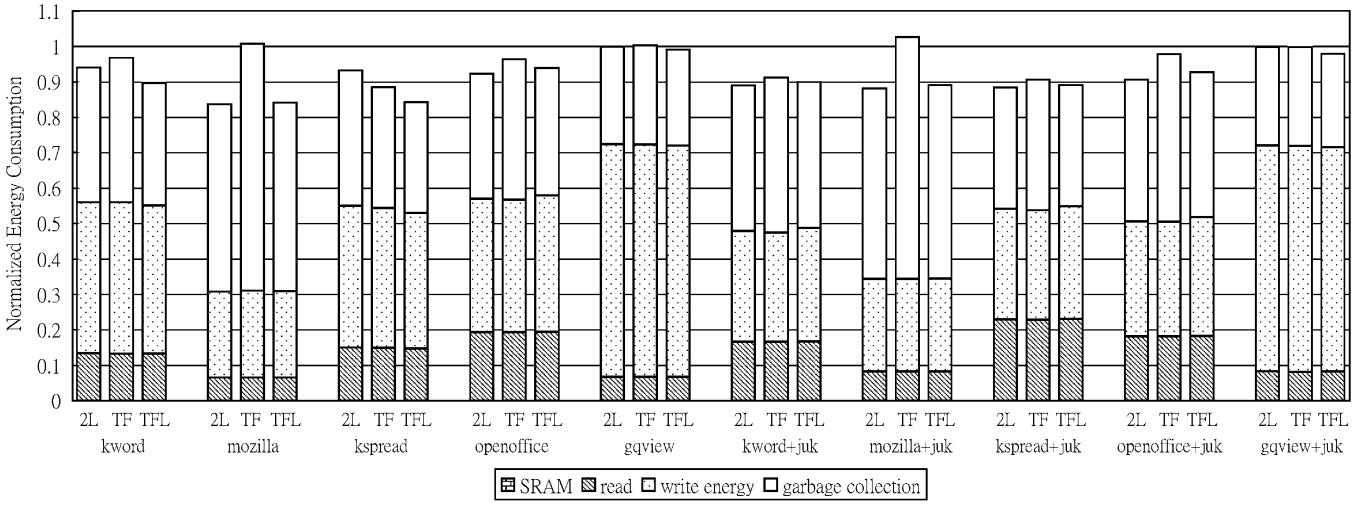


Fig. 14. The average energy consumption under different HotCache schemes (HotCache size: 512 K).

TABLE VII
GARBAGE COLLECTION FREQUENCY AND COPYING OVERHEAD OF MOZILLA+JUK UNDER DIFFERENT HOTCACHE SCHEMES (HOTCACHE SIZE: 512 kB)

	FIFO	LRU	2L	TF	TFL
number of garbage collections	0.93	0.94	0.87	1.03	0.88
garbage collection overhead	0.89	0.91	0.82	1.08	0.85

cache and 3.34% for a 1 MB cache. Although the TFL policy has lower hit rates than TF, it achieves best energy savings. With a 1 MB HotCache, the TFL policy could reduce about 16.3% of flash energy on the average.

One problem particular to the cache management is that higher cache hit rate does not necessarily result in more energy savings. For example, for mozilla+juk, with a 512 kB cache, the TF policy has higher cache hit rate than TFL as shown in Table V (6.20% versus 5.67%), but it achieves less energy savings than TFL. The cause of this abnormal behavior is that the TF policy destroys intrapage locality. This results in higher garbage collection overhead. Fig. 14 shows the energy normalized to the baseline architecture (without the HotCache) for various cache management policies. We break down the energy consumption of the HotCache scheme into four components: garbage collection, write, read and accessing HotCache. We can see that the TF policy has higher garbage collection energy than TFL in almost every application. For mozilla+juk, the TF policy incurs about 13.7% more garbage collection energy than TFL. Table VII shows the normalized garbage collection frequency and average copying overhead incurred by each garbage collection for mozilla+juk. We can see that with the TF policy, the garbage collection frequency is higher than the baseline, and the copying overhead is increased by 8%, while other policies are beneficial for reducing the garbage collection frequency and copying overheads.

C. Duplication-Aware Garbage Collection

To understand the effectiveness of the duplication-aware garbage collection, we report the average number of flash pages of the victim block that exist in the main memory, and the distribution of dirty versus clean pages in Table VIII. Note that those flash pages that are clean at the time when garbage collection occurs, but become dirty later are also counted as dirty pages in Table VIII. For mozilla and kspread, the DA-GC scheme reduces close to half of the live page copying. And more than 70% of these omitted pages are dirty pages. The average garbage collection overhead reduction rate ranges from 17.25% to 54.34%. As mentioned in Section II-C, reducing garbage collection overhead results in less frequent garbage collection. In Table IX, we report the reduction percentage of garbage collection frequency after applying duplication-aware garbage collection. We can see the reduction rates ranges from 7.71% to 53.80%. This leads to significant overall energy reduction as shown in Fig. 15. For mozilla, we see up to 50% of energy reduction. The average energy reduction rate is 24.1%.

D. Analysis of Overall Energy Reduction

Fig. 16 shows the combined effect of HotCache, Subpaging, and DA-GC. In this set of results, we assume 1 MB HotCache managed in the TFL policy, and 512 kB flash pages. We also compare our scheme with the workCFLRU [10], which proposed a new virtual memory replacement policy to reduce writes to flash memory. Since the CFLRU could incur more page faults, we plot the number of page faults of the CFLRU normalized to the baseline architecture in Fig. 16.

The experimental results show that the energy reduction of adopting a 1 MB HotCache, subpaging, and DA-GC together ranges from 9.3% to 75%, and 42.2% on the average. The CFLRU is also quite effective in saving flash memory energy except for applications that have many writes, e.g., gqview, since the CFLRU is not able to find clean pages to replace in this case. The main problem of the CFLRU is its impact on performance. From Fig. 16, we can see that the CFLRU increases the number of page faults significantly for several applications, such as kspread+juk (44.7%). In contrast, our scheme achieves energy savings without causing adverse effect on performance.

Since the three schemes proposed in this paper are orthogonal to the CFLRU, we also evaluate the combined effects of the CFLRU and our scheme in Fig. 16. The results show that using

TABLE VIII
AVERAGE NUMBER OF DUPLICATED PAGES AND GARBAGE COLLECTION OVERHEAD REDUCTION OF DA-GC

application	kword		mozilla		kspread		openoffice		gqview	
% of GC overhead reduction	34.12%		44.26%		54.34%		28.40%		34.18%	
average duplicated pages	2.97		5.02		3.82		2.83		0.85	
	dirty	clean	dirty	clean	dirty	clean	dirty	clean	dirty	clean
	69%	31%	70%	30%	77%	23%	48%	52%	73%	27%
application	kword+juk		mozilla+juk		kspread+juk		openoffice+juk		gqview+juk	
% of GC overhead reduction	30.99%		30.58%		36.33%		20.82%		17.25%	
average duplicated pages	3.45		4.13		4.11		2.57		0.65	
	dirty	clean	dirty	clean	dirty	clean	dirty	clean	dirty	clean
	58%	42%	64%	36%	53%	47%	46%	54%	56%	44%

TABLE IX
GARBAGE COLLECTION FREQUENCY REDUCTION OF DA-GC

application	kword	mozilla	kspread	openoffice	gqview
% GC frequency reduction	24.76%	53.80%	36.05%	18.84%	11.35%
application	kword+juk	mozilla+juk	kspread+juk	openoffice+juk	gqview+juk
% GC frequency reduction	30.34%	43.72%	27.59%	20.72%	7.71%

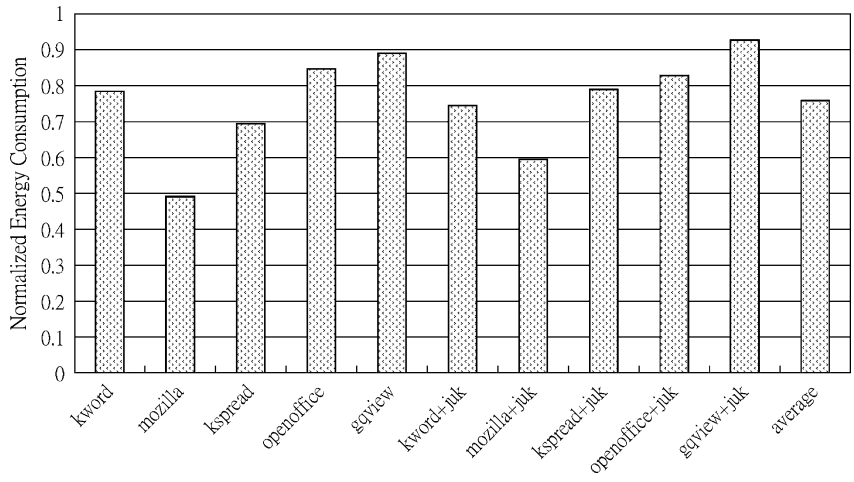


Fig. 15. Overall energy reduction of DA-GC.

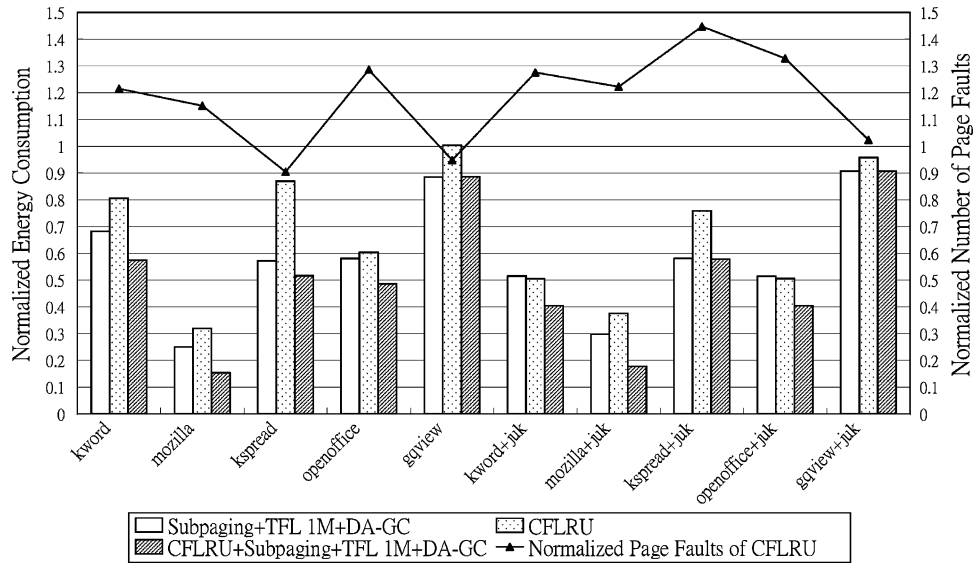


Fig. 16. Combined effect of HotCache, subpaging and DA-GC.

the CFLRU on top of our scheme could further reduce the flash memory energy by 12% at most in mozilla+juk. Therefore, for applications that could trade performance for energy savings, we could use the CFLRU and HotCache/subpaging together to achieve higher energy savings.

VII. Related Work

Previous works combine SRAM with flash memory mainly for performance consideration. Wu *et al.* [21] propose the eNVy system adopting SRAM as write buffers to allow better response time of flash memory and reduce invalidations from write merges. They also propose a locality-gathering garbage collector to achieve even wearing. Park *et al.* [11] propose a NAND XIP architecture applying an SRAM with priority-based caching which application codes with high access frequencies are more likely to be kept in cache. The NAND XIP also use cache prefetching to reduce the access latency to the same level with NOR flash. Both the priority-based caching and prefetching information are gathered through profiling information. Douglis *et al.* [3] first studied the energy consumption issue of flash memory with SRAM write buffer. Their experiments show that SRAM buffering is beneficial for both response time and energy saving.

In the aspect of SRAM caching policies for storage devices, a great number of works were done for hard disk based storage system. For example, Robinson *et al.* [12] propose using frequency-based replacement (FBR) rather than ordinary LRU to gain better performance on disk caches. The FBR maintains reference counts for blocks in the cache, and evicts the block with smallest reference counts among least recently used blocks. Lee, *et al.* [8] propose least recently/frequently used (LRFU) policy which weighting each block with reference count and age to calculate to combined recency and frequency (CRF), and the block with smallest CRF value will be replaced as cache is full. Zhou *et al.* [22] propose multi-queue (MQ) policy, which maintains multiple LRU queues. The policy promote a block to higher level queue when its access frequency arriving some threshold, and demoted least recently used block to lower level queue when the queue is full. Our paper is the first exploring the SRAM caching issues on flash memory based storage system. The TF policy proposed in this paper is simplified from LRFU, and the 2L policy is also an MQ implementation with number of queues is set to two.

In the area of virtual memory system, Park *et al.* [10] propose applying demand paging mechanism as an alternative for traditional shadowing architecture to reduce retention energy of DRAM-based main memory. To achieve better energy saving, they also propose a new replacement policy, CFLRU, to reduce writes to flash memory by keeping dirty pages in memory as long as possible. The subpaging technique [5] is first used to reduce transfer latency in a networked system. This paper is the first to study the effect of subpaging for energy savings of flash memory. Jung *et al.* [6] proposed flash-aware swap system (FASS), which uses page table instead of FTL layer, and using paging information in operating system to identify which page slot is unused to save garbage collection overhead.

Other works on flash memory look at how to reduce garbage collection overheads and increase I/O access parallelism. Conventional greedy garbage collection policy cannot avoid recycling frequently modified data, which may be invalidated soon after, and results in more garbage collection overhead. Addressing on the problem, Rosenblum *et al.* [13] propose a cost-benefit garbage collection policy using value-based heuristic considering both reclaiming overhead and data update frequency for log-structured file systems. Kawaguchi *et al.* [7] modified the calculation cost-benefit values for flash memory systems. Instead of calculating the cost-benefit value as $(\text{age} \times (1-u))/(1+u)$, Kawaguchi *et al.* choose to reclaim the data segment with largest $(\text{age} \times (1-u))/2u$ due to the different writing costs for hard disk and flash memory. Chiang *et al.* [2] further modified cost-benefit garbage collector with the calculation of erase count per block to achieve evenly wearing. Chiang's work also keeps the update count of each LBA for identifying hot data, and clusters hot data together to further reduce garbage collection overhead. Chang *et al.* [1] propose the adaptive stripping architecture to exploit the I/O parallelism from multibanked flash memory system by using dynamic bank assignment policy rather than static policies. They also propose the use of hot–cold data separation for reducing garbage collection overhead and balancing the bank access frequencies. Rather than the hot–cold separation method used in the above work, Chang *et al.* designed a low cost and efficient hot data identification method by maintaining two LRU lists. Addressing on the energy consumption issue of multibanked flash storage systems, Wu *et al.* [20] present an architecture that supports programmed I/O to prevent processor from wasting cycles. By reordering the I/O requests, the energy for switching among power states can also be reduced.

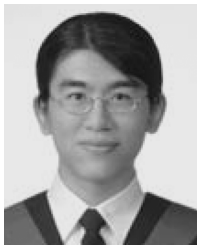
VIII. Conclusion

In this paper, we propose three energy-efficient techniques for flash memory management in the virtual memory system. The subpaging technique divides a main memory page into a set of subpages in the granularity of flash page size. On a page fault, only dirty subpages are written back to flash memory instead of a full main memory page as in the conventional virtual memory system. The experimental results show the subpaging technique reduces about 15.8% of flash memory energy on the average, and 21.2% for multiprogramming workloads. The HotCache scheme stores frequent writes to reduce flash energy. We find that higher HotCache hit rate does not necessarily lead to higher flash energy savings. The intrapage locality needs to be preserved when writing data from the storage buffer to flash memory. The TFL policy can achieve about 16.3% of energy saving. DA-GC exploits data redundance between the main memory and flash memory to eliminate unnecessary live page copying in garbage collection. The experimental results show that DA-GC achieves up to 51% energy reduction. Joint use of subpaging, HotCache, and DA-GC can reduce 42.2% of flash memory energy on average.

References

[1] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proc. 8th IEEE Real-Time and Embedded Technol. Appl. Symp.*, Sep. 2002, pp. 24–27.

[2] M. Chiang, P. Lee, and R. Chang, "Managing flash memory in personal communication devices," in *Proc. 1997 Int. Symp. Consumer Electronics (ISCE'97)*, Singapore, Dec. 1997, pp. 177–182.
[3] F. Douglis, F. Kaashoek, B. Marsh, R. Caceres, K. Li, and J. Tauber, "Storage alternatives for mobile computers," in *Proc. 1994 Symp. Operating Syst. Design Implementation*, Nov. 1994, pp. 25–37.
[4] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "The design of DEETM: A framework for dynamic energy efficiency and temperature management," *J. Instruction-Level Parallelism*, vol. 3, 2002.
[5] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. Evans, A. R. Karlin, H. M. Levy, and M. K. Vernon, "Reducing network latency using subpages in a global memory environment," in *Proc. 7th ACM Conf. Archit. Support Program. Languages Operating Syst.*, 1996, pp. 258–267.
[6] D. Jung, J.-S. Kim, S.-Y. Park, J.-U. Kang, and J. Lee, "Fass: A flash-aware swap system," in *Proc. Int. Workshop Software Support for Portable Storage*, Mar. 2005.
[7] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. 1995 USENIX Tech. Conf.*, Jan. 1995, pp. 155–164.
[8] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," *Meas. Modeling Comput. Syst.*, pp. 134–143, 1999.
[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.
[10] C. Park, J.-U. Kang, S.-Y. Park, and J.-S. Kim, "Energy-aware demand paging on NAND flash-based embedded storages," in *ISLPED '04: Proc. IEEE/ACM Int. Symp. Low Power Electron. Design*, pp. 338–343.
[11] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim, "A low-cost memory architecture with NAND XIP for mobile embedded systems," in *Proc. 1st IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2003, pp. 138–143.
[12] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst.*, May 1990, pp. 134–142.
[13] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," in *Proc. 13th Symp. Operating System Principles*, October 1991, pp. 1–15.
[14] Samsung Electronics CO., LTD, Datasheet of Samsung K9F1208R0B NAND Flash 2004.
[15] Samsung Electronics CO., LTD, Datasheet of Samsung OneNAND128 2004.
[16] Samsung Electronics CO., LTD, Datasheet of Samsung K9K2G08X0A NAND Flash 2006.
[17] P. Shivakumar and N. P. Jouppi, CACTI 3.0: An Integrated Cache Timing, Power and Area Model Compaq Comput. Corp., Aug. 2001, Tech. Rep..
[18] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. 2002 Design, Automation Test Eur. Conf. Exhib.*, Mar. 2002, pp. 409–417.
[19] H.-W. Tseng, H.-L. Li, and C.-L. Yang, "An energy-efficient virtual memory system with flash memory as the secondary storage," in *ISLPED '06: Proc. IEEE/ACM Int. Symp. Low Power Electron. Design*, pp. 418–423.
[20] C.-H. Wu, T.-W. Kuo, and C.-L. Yang, "Energy-efficient flash memory storage systems with an interrupt emulation mechanism," in *CODES + ISSS 2004: Proc. IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synthesis*, pp. 134–139.
[21] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, Oct. 1994, pp. 86–97.
[22] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. General Track: 2002 USENIX Annu. Tech. Conf.*, 2001, pp. 91–104.



Han-Lin Li received the B.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C., in 2005, where he is currently working toward the Ph.D. degree.

His research interests include energy-efficient design and flash memory.

Chia-Lin Yang (M'02) received the B.S. degree from the National Taiwan Normal University, Taiwan, R.O.C., in 1989, the M.S. degree from the University of Texas at Austin in 1992, and the Ph.D. degree from the Department of Computer Science, Duke University, Durham, NC, in 2001.

In 1993, she joined VLSI Technology Inc. (now Philips Semiconductors) as a Software Engineer. She is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan. Her research interests include energy-efficient microarchitectures, memory hierarchy design, and multimedia workload characterization.

Dr. Yang is the recipient of a 2000–2001 Intel Foundation Graduate Fellowship Award and 2005 IBM Faculty Award.

Hung-Wei Tseng received the B.S. and M.S. degrees from the Department of Computer Science and Information Engineering at National Taiwan University, Taipei, in 2003 and 2005, respectively. He is currently working toward the Ph.D. degree in Department of Computer Science and Engineering at the University of California, San Diego.

His research interests include energy-efficient computer system design and multicore processors.