CFLRU: A Replacement Algorithm for Flash Memory*

Seon-yeong Park[1], Dawoon Jung[1], Jeong-uk Kang[1], Jin-soo Kim[2], and Joonwon Lee[2]

Division of Computer Science,
Korea Advanced Institute of Science and Technology (KAIST)
373-1 Guseong-dong, Daejeon, Korea

[1]{parksy, dwjung, ux}@calab.kaist.ac.kr
[2]{jinsoo, joon}@cs.kaist.ac.kr

ABSTRACT

In most operating systems which are customized for disk-based storage system, the replacement algorithm concerns only the number of memory hits. However, flash memory has different read and write cost in the aspects of time and energy so the replacement algorithm with flash memory should consider not only the hit count but also the replacement cost caused by selecting dirty victims. The replacement cost of dirty page is higher than that of clean page with regard to both access time and energy consumption. In this paper, we propose the Clean-First LRU (CFLRU) replacement algorithm that exploits the characteristics of flash memory. CFLRU splits the LRU list into the working region and the clean-first region and adopts a policy that evicts clean pages preferentially in the clean-first region until the number of page hits in the working region is preserved in a suitable level. Using the trace-driven simulation, the proposed algorithm reduces the average replacement cost by 28.4% in swap system and by 26.2% in buffer cache, compared with LRU algorithm. We also implement the CFLRU algorithm in the Linux kernel and present some optimization issues.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage management-*Secondary storage*

General Terms

Algorithms, Design, Measurement, Performance.

Keywords

Flash Memory, Replacement Algorithm, Embedded Storage.

* This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

CASES'06, October 23–25, 2006, Seoul, Korea.

1. INTRODUCTION

Flash memory has been gaining popularity in mobile embedded systems as non-volatile storage due to its characteristics such as small and lightweight form factor, solid-state reliability, and low power consumption [1]. The emergence of single flash memory chip with several gigabytes capacity makes a strong tendency to replace magnetic disk with flash memory for the secondary storage of mobile computing devices such as tablet PCs, PDAs, and smart phones [2, 3].

For decades, traditional operating systems have been optimized in various ways assuming that the secondary storage consists of magnetic disks. Unfortunately, flash memory exhibits different characteristics compared to magnetic disks. Most notably, flash memory does not have a seek time, and its data can not be overwritten before being erased. In addition, flash memory has asymmetric read and write operation characteristics in terms of performance and energy consumption. Therefore, it is necessary to revisit various operating system policies and mechanisms, and to optimize them for flash memory-based secondary storage.

In this paper, we focus on a cache replacement policy for embedded systems equipped with flash memory as secondary storage. Most operating systems use an approximated LRU (Least Recently Used) algorithm for a replacement policy. However, operating systems with flash memory need to adopt a new replacement policy which considers not only the cache hit rate but also the replacement cost. The replacement cost occurs when a dirty page which is modified before evicting from page cache is written to flash memory to reclaim free space. The write operations to flash memory necessitate more time and energy than read operations, and moreover increasing the number of write operations will accompany even more costly erase operations. Therefore, the replacement policy should reduce the number of write and erase operations on flash memory, while avoiding escalation of memory misses which might lead to a large number of read operations.

We propose a new replacement policy, called Clean-First LRU (CFLRU), which takes into consideration of the imbalance of read and write costs of flash memory when replacing pages. The basic idea behind CFLRU is to keep a certain amount of dirty pages deliberately in page cache to reduce the number of flash write operations, while preventing the overall performance from being significantly affected due to the degraded cache hit rate.

We simulated CFLRU with two kinds of traces; one is virtual memory traces to evaluate the effect of the replacement algorithm in swap system, and the other is block access traces to evaluate it

in a file system buffer cache. Using these traces, we find that CFLRU algorithm reduces the average replacement cost by 28.4% in swap system and by 26.2% in buffer cache, when comparing with LRU. When calculating the replacement cost, we assume that the cost of a write operation is eight times higher than that of a read operation. We have also implemented our algorithm in the Linux kernel and presented some optimization issues for flash memory.

The rest of this paper is organized as follows. Section 2 presents the characteristics of flash memory and the motivation of our work. Section 3 describes our CFLRU replacement algorithm. Section 4 shows the simulation methodology and simulated results with real workload traces. In section 5, we apply our algorithm to the Linux kernel and section 6 shows the implementation results. Section 7 overviews the related work. Finally, we conclude in section 8.

2. BACKGROUND AND MOTIVATION

2.1 Flash Memory

There are two different types of flash memory: NAND flash and NOR flash. Major difference between them is the bus interface [4]. NAND flash has an I/O interface with control inputs, but NOR flash has directly connected address and data buses similar to SRAM. NAND flash is especially appropriate for data storage because of higher density, lower cost, and faster write operations. NOR flash is designed for code storage and execute-in-place (XIP) applications due to its faster read operations and random access capabilities.

The characteristics of flash memory are significantly different from magnetic disks. First, flash memory has no latency associated with the mechanical head movement to locate the proper position to read or write data. In magnetic disks, this seek time has been one of the most time-consuming parts in I/O activity, and operating systems have performed various optimizations such as prefetching and disk scheduling to amortize the cost.

Second, flash memory has asymmetric read and write operation characteristics in terms of performance and energy consumption. Table 1 compares the access time and the energy consumption in flash memory when 4KB data is read, written, or erased. The time and energy values are extracted from [21]. From table 1, we can see that the access time and the energy consumption of a write operation are about 6.3 ~ 6.4 times higher than those of a read operation in NAND flash memory, and the ratio becomes much larger in NOR flash memory.

Third, flash memory does not support in-place update; the write to the same page can not be done before the page is erased. Thus, as the number of write operations increases so does the number of erase operations. If the erase operations are involved, the cost imbalance would be even worse. When we consider the potential erase cost usually accompanied with flash write operations, performing a flash write operation costs more than 8 times higher than performing a flash read operation.

Finally, blocks of flash memory are worn out after the specified number of write/erase operations. Therefore, flash memory requires a well-designed garbage collection scheme to evenly wear out the flash memory region.

Table 1. The characteristics of flash memory [21]

NAND	Access Time *(us/4KB)*	Read	284.2
		Write	1833.0
		Erase	499.2
	Energy Consumption *(uJ/4KB)*	Read	9.4
		Write	59.6
		Erase	16.5
NOR	Access Time *(us/4KB)*	Read	53.8
		Write	14054.4
		Erase	15616.0
	Energy Consumption *(uJ/4KB)*	Read	8.6
		Write	3251.2
		Erase	3609.6

2.2 Motivation

One of the target applications of flash memory is embedded storage of mobile devices such as MP3 players, PDAs (Personal Digital Assistants), PMPs (Portable Media Players), and smart phones. In these devices, the operating system has much more chances to optimize I/O subsystem to adopt specialized features based on the characteristics of flash memory. Using the kernel-level information effectively, the operating system can reduce the number of costly write and erase operations, and as a result, increase the performance [5].

Our work is motivated by the observation of replacement policies of the existing operating systems. Most operating systems are customized for disk-based storage system and their replacement policies only concern the number of cache hits. However, operating systems that use flash memory as secondary storage should consider different read and write cost of flash memory when they replace pages to reclaim free space. In this paper, we propose the Clean-First LRU (CFLRU) replacement algorithm for flash memory. CFLRU tries to reduce the number of costly write operations and potential erase operations until the degradation of cache hit rate does not harm the performance. To our best knowledge, CFLRU is the first replacement algorithm proposed for flash memory.

3. CFLRU ALGORITHM

3.1 Overview

When cache replacement occurs, two kinds of replacement costs are involved. One is generated when a requested page is fetched from secondary storage to the page cache in RAM. Using Belady's MIN algorithm [8], this cost can be minimized by selecting a victim that has the largest forward distance in the future references. Among online algorithms, LRU has been commonly used for replacement algorithm because it exploits the property of locality in references. The other cost is generated when a page is evicted from the page cache to secondary storage, i.e., flash memory. This cost can be minimized by selecting a clean page for eviction. A clean page contains the same copy of the original data in flash memory thus the clean page can be just dropped from the page cache when it is evicted by the replacement policy.

Satisfying only one kind of replacement cost would benefit from its advantage, but for a long term, it would affect the other kind of replacement cost, and vice versa. For example, a replacement

policy might decide to keep dirty pages in cache as many as possible to save the write cost on flash memory. However, by doing this, the cache will run out of space, and consequently the number of cache misses will be increased dramatically, which, in turn, will increase the replacement cost of reading requested pages from flash memory. On the other hand, a replacement policy that focuses mainly on increasing the cache hit count will evict dirty pages, which will increase the replacement cost of writing evicted pages into flash memory. Thus, a sophisticated scheme to compromise both sides of efforts is needed to minimize the total cost.

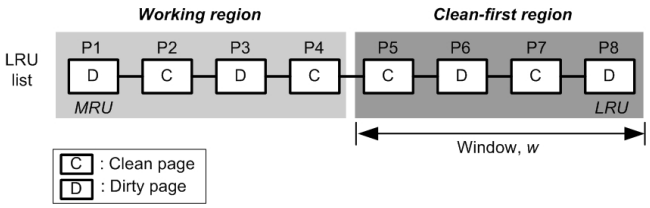


Figure 1. Example of CFLRU algorithm

For this purpose, we propose a new replacement algorithm, called CFLRU (Clean-First LRU), which is modified from the LRU algorithm. CFLRU divides the LRU list into two regions to find a minimal cost point, as shown in Figure 1. The working region consists of recently used pages and most of cache hits are generated in this region. The clean-first region consists of pages which are candidates for eviction. CFLRU selects a clean page to evict in the clean-first region first to save flash write cost. If there is no clean page in this region, a dirty page at the end of the LRU list is evicted. For example, under the LRU replacement algorithm, the last page in the LRU list is always evicted first. Thus, the priority for being a victim page is in the order of P8, P7, P6, and P5, in Fig. 2. However, under the CFLRU replacement algorithm, it is in the order of P7, P5, P8, and P6.

The size of the clean-first region is called a window size, w. It is important to adjust the window size properly, because the hit rate may fall dramatically if the window size grows too large. This issue will be examined in the next subsection.

3.2 Window Size for Minimal Replacement Cost

Finding the right window size of the clean-first region is important to minimize the total replacement cost. A large window size will increase the cache miss rate and a small window size will increase the number of evicted dirty pages, that is, the number of flash write operations. The flash write operations can also cause a large number of the costly erase operations, as mentioned in section 2.1. Therefore, the window size of the clean-first region needs to be decided properly in order to minimize the overall replacement cost.

Let us assume that C_W is the cost of a flash write operation and C_R is the cost of a flash read operation. If N_D denotes the number of dirty pages that should have been evicted in the LRU order but are kept in the cache, the benefit of the CFLRU algorithm is $C_W \bullet N_D$. If N_C is the number of clean pages that are evicted instead of dirty pages within the clean-first region, and $P_i(k)$ is the probability of the future reference of a clean page, i, which is evicted at the k-th position, the cost of the CFLRU algorithm can be calculated by $\sum_{i}^{N_C} C_R \bullet P_i(k)$. Figure 2 shows the probability of the future reference at each position of the LRU list. In this graph, the x-axis indicates the position of a page in the LRU list. The left-end of the x-axis is the most recently used page and the right-end of the x-axis the least recently used page. The y-axis indicates the probability of the future reference at each position of the pages. For example, if the k-th page in the LRU list is selected for an eviction, it is likely to be referenced in the future and fetched into the cache again with its probability of $P(k)$. From what has been discussed above, formula (1) summarizes the proper window size, w, of the clean-first region.

$$\underset{w}{MAX}\left[C_W \bullet N_D - \sum_{i}^{N_C} C_R \bullet P_i(k)\right] \quad (1)$$

However, in the real world, it is not easy to find the probability of the future reference at each position of the LRU list. In this paper, we investigate the proper window size of the clean-first region with statically defined parameters and also devise a method to adjust the window size dynamically. The static method initially fixes the window size with an average well-performed value obtained from repetitive experiments over a predetermined application set. However, static method can not dynamically adjust the window size to various cache size and different application sets. The dynamic method can properly adjust the window size based on periodically collected information about flash read and write operations. If v_W and v_R represent the ratio of write and read operations for a given time period, respectively, the cost difference between adjacent periods, $\Delta(v_W \cdot C_W + v_R \cdot C_R)$ can control whether to enlarge or reduce the window size. With our experiments, the dynamic algorithm performs well with various cache states imposed by different application sets.

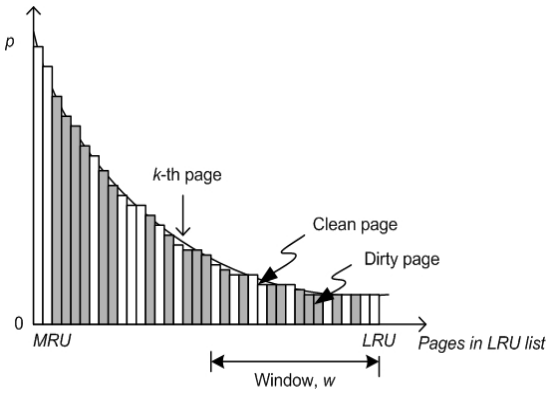


Figure 2. Window size of CFLRU algorithm

4. SIMULATION STUDY

Before implementing the CFLRU replacement algorithm in the Linux kernel, we performed a simulation study. The objective of this study is to compare the performance of the pure LRU with the

CFLRU algorithm. In addition, we simulated the offline algorithm, Belady's MIN. It is known to be the optimal algorithm that maximizes the cache hit rate but is not a cost optimal algorithm.

4.1 Simulation methodology

Our simulation is performed with two different types of real workload traces. One is the trace for swap system and the other is for file system. For swap system, we gathered virtual memory reference traces using a profiling tool, called Valgrind [9]. We have slightly modified Valgrind's cache simulation module called Cachegrind to obtain instruction and data reference traces. For file system, we gathered block reference traces directly from the buffer cache under the ext2 file system. We chose five different applications and executed them on Linux/x86 machine, according to the scenario shown in Table 2. The characteristics of the two kinds of traces used in simulations are described in Table 3.

In the simulations, we assume that the system has 32MB SDRAM and a 128MB flash memory for swap space or file system. It is also assumed that the operating system and X-windows system consume about 16MB of SDRAM, thus each application can freely use about 16MB of the remaining SDRAM for cache space. Because the Valgrind profiling tool can gather virtual memory reference traces only for one application at a time, the cache size is dedicated to one application.

Table 2. Scenarios used in gathering traces

Workloads	Scenarios
Gqview	Performs a slide show of seven images and adjusts their sizes.
Kword	Edits several lines and saves in a text document.
Kspread	Calculates sum, avg.,min., and max. of numerical data, and sorts them.
Acrobat reader	Views a PDF document.
Mozilla	Browses several web sites such as shopping mall, news center, mail server, etc.

We compare the performance of Belady's MIN algorithm, LRU, CFLRU with static window size (CFLRU-static), and CFLRU with dynamic window size (CFLRU-dynamic). To evaluate the replacement cost, we use the weighted count of read and write operations on flash memory. The write count is weighted with eight times higher than the read count. This is based on the access time and the energy consumption characteristics of NAND flash memory taking into account the cost of potential erase operations (cf. Table 1). Note that as discussed in section 2.1, the relative cost of write operation is increased by more than an order of magnitude in NOR flash memory.

Table 3. Workloads used in simulations

Workloads	Memory used (MB)	Memory references			
		Total	Instruction	Data	
			Read	Read	Write
Gqview	21.49	28,940,881	896,355 (3.1 %)	11,437,768 (39.5 %)	16,606,758 (57.4 %)
Kword	28.57	4,779,386	1,025,217 (21.4 %)	2,837,916 (59.4 %)	916,253 (19.2 %)
Kspread	40.24	11,366,261	1,515,521 (13.3 %)	6,264,476 (55.1 %)	3,586,264 (31.6%)
Acrobat reader	37.28	10,815,848	2,062,732 (19.1 %)	2,256,161 (20.9 %)	6,496,955 (60.0 %)
Mozilla	39.64	41,533,372	20,243,704 (48.7 %)	14,893,383 (35.9 %)	6,396,285 (15.4 %)

(a) Virtual memory references

Workloads	Buffer used (MB)	Buffer block references		
		Total	Read	Write
Gqview	21.16	84,226	82,792 (98.3 %)	1,434 (1.7 %)
Kword	34.71	33,614	29,402 (87.5 %)	4,212 (12.5 %)
Kspread	33.63	34,719	33,115 (95.4 %)	1,604 (4.6 %)
Acrobat reader	28.86	37,884	36,244 (95.7 %)	1,640 (4.3 %)
Mozilla	35.89	39,141	25,557 (65.3 %)	13,584 (34.7 %)

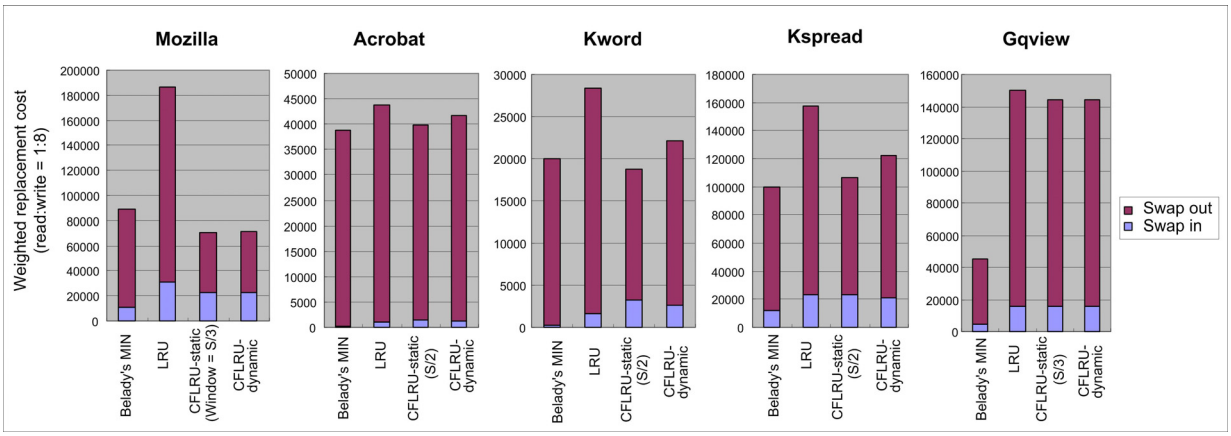
(b) Buffer block references

4.2 Simulation Results

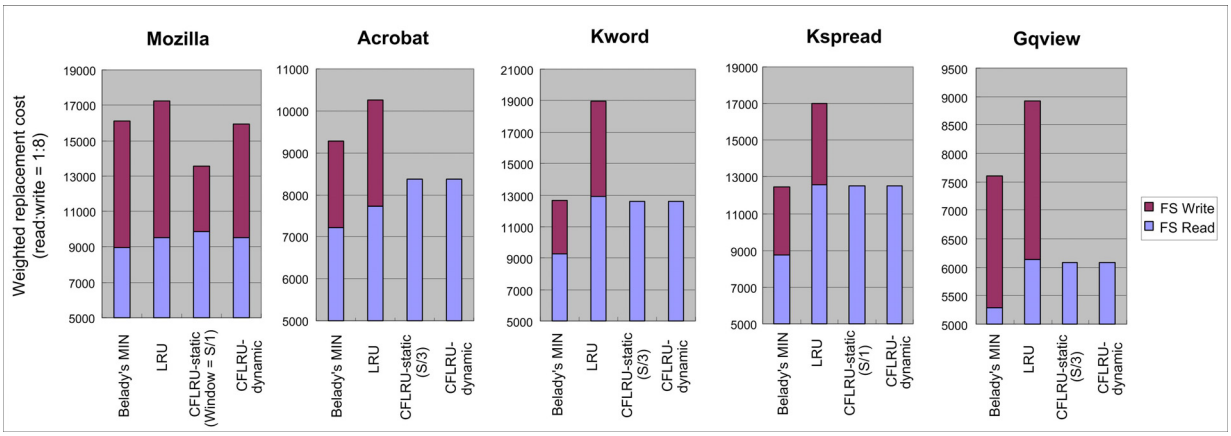
Figure 4 shows the replacement cost of four algorithms. In these results, it is clear that Belady's MIN algorithm is not optimal with flash memory. However, it provides the performance by minimizing the hit count. CFLRU-static shown in the graph presents the lowest cost among six different configurations of the window size, in the form of S/x, where S is the cache size and x varies from 1 to 6. Compared with the LRU algorithm, the average replacement costs of CFLRU-static and CFLRU-dynamic are reduced by 28.4% and by 23.1%, respectively, when they are used for swap system. In the case of file system buffer cache, the average replacement costs of CFLRU-static and CFLRU-dynamic are reduced by 26.2% and by 23.5%, respectively. Sometimes, the dynamic algorithm has a small amount of performance degradation when comparing with the static algorithm. This is due to a wrong decision that might happen if the computed replacement cost to predict the future events is not accurate with the actually occurred events. However, CFLRU-dynamic still has reasonable performance, and it has an advantage that we do not have to reconfigure the window size whenever the cache size and the application set change.

5. IMPLEMENTATION ON LINUX

We have implemented the CFLRU replacement algorithm based on the Linux kernel 2.4. In this section, we briefly address the replacement policy of the Linux kernel, and explain our modification for the CFLRU replacement algorithm and some optimization issues.



(a) Replacement cost for swap system



(b) Replacement cost for file system buffer cache

Figure 4. Replacement costs of Belady's MIN, LRU, CFLRU-static, and CFLRU-dynamic

5.1 Original Linux Page Reclamation

The Linux kernel 2.4 has the page cache that consists of two pseudo-LRU lists, the active list and the inactive list, as shown in Figure 5 [10]. The pages in the active list are recently accessed while the pages in the inactive list are not. When the kernel decides to make free space, it starts the reclaiming phase. First, it scans the pages in the inactive list. Priority value decides the scanning range of pages in the inactive list. Priority of the inactive list is increased, from 1/6 to 1/5, …, and 1, until the enough number of pages (usually 32) is freed. Second, if there are too many process-mapped pages, it starts the swap-out phase. Page reclamation in the swap-out phase is performed in a round-robin fashion in the Linux kernel 2.4, starting the scanning from the memory that was last checked. In this phase, the pages that are in the active list or recently accessed are skipped.

5.2 CFLRU Implementation

To implement the CFLRU replacement algorithm in the Linux kernel, we insert an additional reclamation function that chooses clean victims first before starting the original reclamation phases. Similar to the original Linux kernel, the additional reclamation function consists of two phases. In the first phase, clean pages in the inactive list are evicted first until the enough number of pages is freed. In the original Linux kernel, dirty pages become ready to write when they are selected as victims, but under our CFLRU, dirty pages are simply skipped. In the second phase, clean pages that belong to the process region are swapped out. As mentioned above, reclamation in the swap-out phase is not based on LRU in the Linux kernel 2.4. It is future work to modify the reclamation algorithm of swap system in the CFLRU fashion.

In the inactive list of the Linux kernel, the concept of priority is correctly matched with that of the window size of the clean-first region in CFLRU. We configure the priority value statically, as default. However, the window size for the minimal replacement cost depends on the specific reference patterns of applications. To adjust the window size properly, the priority values can be changed dynamically. A kernel daemon periodically checks the replacement cost and compare with last replacement cost to decide whether to increase or decrease the priority. To avoid oscillation of the priority value, the difference between the current cost and the last cost has to exceed a predefined threshold.

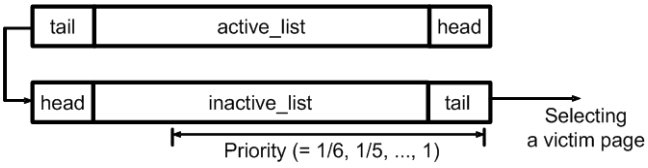


Figure 5. LRU lists in Linux

5.3 Optimizations for flash memory

The I/O subsystem of the Linux kernel is optimized for disk-based storage. Disk scheduling policy and sequential read-ahead are examples of optimizations to reduce the seek time of disk-based storage. However, the disk optimization does not help improve the performance of a system with flash memory. It might rather decrease the performance due to the wasting time for doing the disk optimizations. Especially, read-ahead has a bad effect on cache hit rate when most of read-ahead pages are never accessed in the near future. The cache space for actively used pages is reduced because of read-ahead pages. In our experience, sequential read-ahead pages achieve low hit rate in many cases. Random access time of flash memory is faster than that of magnetic disk because flash memory has no mechanical parts to search data. We remove the sequential read-ahead function in the Linux kernel and these results in large improvement on cache hit rate.

6. IMPLEMENTATION RESULTS

The CFLRU replacement algorithm is evaluated on a system with a Pentium IV processor and 32MB SDRAM running the Linux kernel 2.4.28. We evaluate the performance of the replacement algorithms per application, so that the choice of the cache size, 32MB, is appropriate for the cache replacement algorithm. The system has been emulated to have 64MB and 256MB of flash memory for swap space and for file system (ext2), respectively. The emulated flash memory has the same latency as real NAND flash memory [21].

We compare four Linux kernel implementations; plain Linux kernel, Linux kernel without swap read-ahead (no-readahead), Linux kernel with CFLRU static algorithm (CFLRU-static), and Linux kernel with CFLRU dynamic algorithm (CFLRU-dynamic). The Linux kernels with CFLRU static algorithm and CFLRU dynamic algorithm also do not exploit read-ahead method. The window size of the CFLRU static algorithm is 1/4 of the inactive list. To measure the performance of four methods, we chose five applications; gcc, tar, diff, encoding, and file system benchmarks. These applications are not generally used in mobile devices but are proper for measuring the time delay.

Figure 6 shows the weighted cost calculated by the number of flash read and write operations. As mentioned before, flash memory is partitioned into two regions; one for swap system and the other for file system. We measured the total number of flash read and write operations and weighted the write count with eight times higher than read count. The no read-ahead method reduces the overall read count when comparing the plain Linux kernel. In CFLRU methods, the overall flash read count is slightly increased, while the flash write count is significantly decreased, thus the sum of weighed counts is reduced when comparing to the no read-ahead method. Figure 7 shows the normalized time delay and expected energy. The expected energy is calculated with byte read and byte write counts from/to flash memory. The average time delay of no read-ahead method is reduced by 2.4%, and the expected energy saving by 4.4%. The average time delays of CFLRU-static and CFLRU-dynamic method are reduced by 6.2% and by 5.7%, respectively, while the expected energy savings of CFLRU-static and CFLRU-dynamic by 11.4% and 12.1%, respectively.

7. RELATED WORK

There have been plenty of studies for cache replacement algorithms. Belady's MIN algorithm [8] is known for offline optimal and it provides the lower bound of miss counts compared to other online algorithms. In online algorithms, the LRU [11, 12] algorithm is widely used. Many operating systems use an approximated LRU for cache management. To enhance the performance over the LRU algorithm for particular workloads, various algorithms such as 2Q [13], LRU-K [14], EELRU [15], LRFU [16], and ARC [17] have been proposed. However, all these algorithms pay attention only to the minimization of cache miss counts.

Recently, several studies have focused on the replacement algorithm with various miss costs, especially, under assumption that all pages have a uniform size. The GreedyDual [18] algorithm considers multiple miss costs, and when replacement occurs, the costs of all blocks are reduced by the cost of an evicting block. According to the research of Jeong and Dubois [20], the GreedyDual algorithm does not perform well with a page replacement algorithm when the difference between low and high cost is small. Jeong and Dubois proposed a cost-sensitive algorithm in CC-NUMA multiprocessor environment [19, 20]. They first devised an offline algorithm for two different miss costs [19] and later proposed an online algorithm based on LRU [20]. The online cost-sensitive algorithm selects a victim whose cost is lower than the cost of the least recently used block. The cost of the least recently used block is reduced by twice the amount of the miss cost of the victim.

These algorithms only consider the replacement cost that occurs when pages or blocks are fetched into the cache. In addition, they do not consider the potential hit-ratio of victims that are evicted instead of high-cost blocks. In contrary, CFLRU counts the replacement cost both evicting pages from cache and fetching pages from flash memory, and pages with high hit ratio are kept in cache with the expectation of reducing flash read costs.

8. CONCLUSION

This paper presents the Clean-First LRU (CFLRU) replacement algorithm for flash memory. CFLRU considers the fact that flash memory has asymmetric read and write cost. CFLRU tries to reduce the number of costly write and potential erase operations as long as the degradation of cache hit rate does not harm the performance. We configure the window size of the clean-first region for both statically and dynamically. The dynamic CFLRU algorithm has a benefit that we do not have to reconfigure the window size each time the memory size changes, while achieving the similar performance results with the static CFLRU algorithm configured with the best performed window size. We simulated our algorithm with real workload traces and implemented it on the

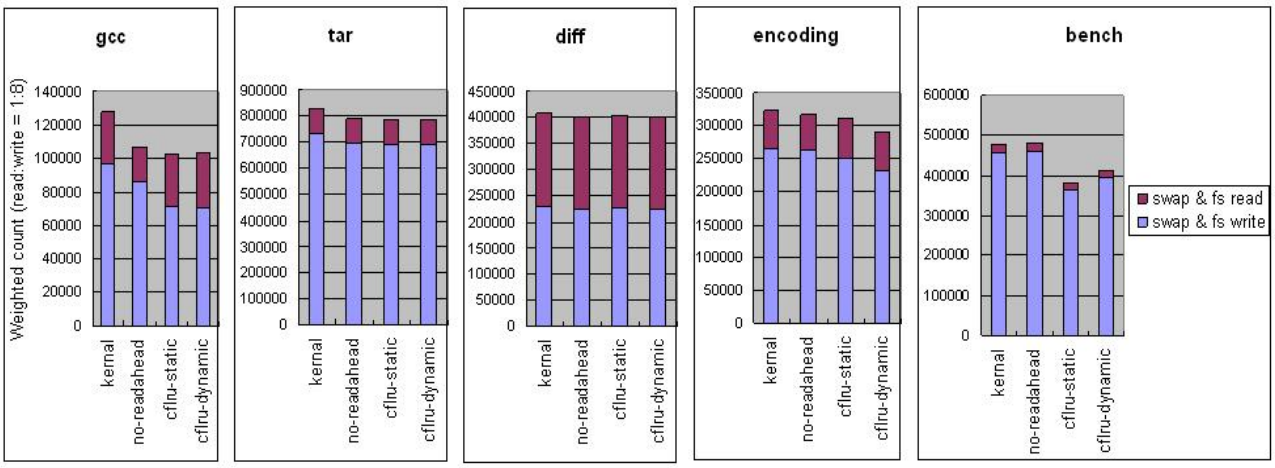


Figure 6. Weighted flash read and write counts under swap and file system cache

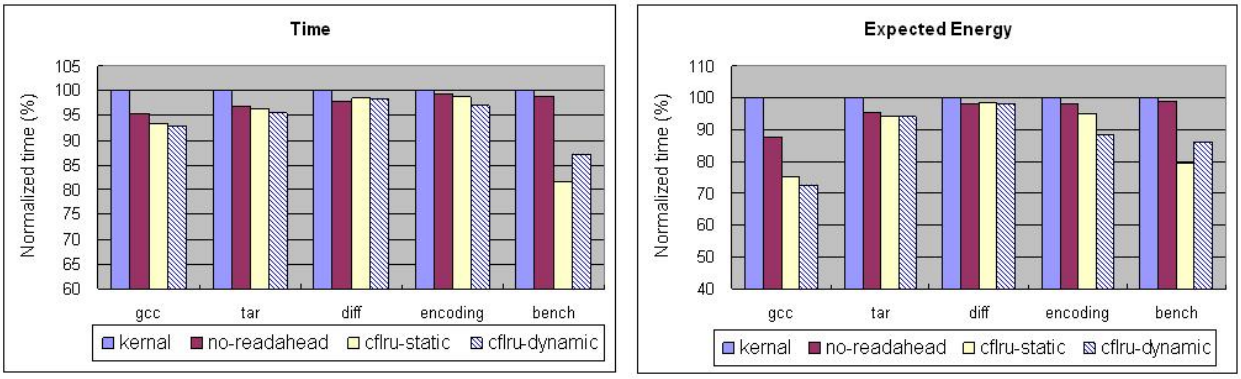


Figure 7. Time delay and expected energy

Linux kernel. In addition, we applied some optimizations for flash memory to the Linux kernel.

Our simulation results show that, in swap system, static and dynamic methods of CFLRU reduce the replacement cost by 28.4% and by 23.1%, respectively, in comparison to LRU. In file system buffer cache, static and dynamic methods reduce the replacement cost by 26.2% and 23.5%, respectively. We also evaluate the CFLRU implementation on Linux and compare the performance of the original Linux kernel, the Linux kernel without read-ahead method, and CFLRU without read-ahead method. The average time delay of no read-ahead method is reduced by 2.4%, and the expected energy saving by 4.4%. The average time delays of CFLRU-static and CFLRU-dynamic method are reduced by 6.2% and by 5.7%, respectively, while the expected energy savings of CFLRU-static and CFLRU-dynamic by 11.4% and 12.1%, respectively.

One minor issue in CFLRU algorithm is that keeping dirty pages in the SDRAM cache may cause a problem when the power has been shut down abruptly. However, this is not a unique problem in CFLRU. The widely used replacement algorithms such as LRU also have the same problem. Although the amount of lost dirty blocks may become slighted larger in case of CFLRU, it is unnecessary to concern in swap system because the process data in swap system need not recover after crash or shut-down. In the case of file system buffer cache, dirty blocks can expose a recovery problem. To cope with this problem, we need to use the CFLRU algorithm with journaling file system.

9. REFERENCES

[1] R. Cáceres, F. Douglis, K. Li, and B. Marsh, "Operating System Implications of Solid-State Mobile Computers," Proc. of the 4th IEEE Workshop on Workstation Operating Systems, October 1993.

[2] Samsung Flash memory, http://www.samsung.com/Products/Semiconductor/Flash/.

[3] M-systems Fast Flash Disks, http://www.m-sys.com/site/en-US/Products/IDESCSIFFD/IDESCSIFFD.

[4] Memory Technology Devices, http://www.linux-mtd.infradead.org/doc/nand.html.

[5] D. Jung, J. Kim, S. Park, J. Kang, and J. Lee, "FASS: A Flash-Aware Swap System", Proc. of International Workshop on Software Support for Portable Storage (IWSSPS), Mar. 2005.

[6] Yet Another Flash Filing System (YAFFS), Aleph One Company.

[7] D. Woodhouse, “JFFS: The Journaling Flash File System”, Proc. of Ottawa Linux Symposium, 2001.

[8] L. Belady, “A Study of Replacement Algorithms for a Virtual-Storage Computer,” IBM Systems Journal, vol.5, no.2, pp.78-101, 1966.

[9] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” Electronic Notes in Theoretical Computer Science, vol.89, no.2, 2003.

[10] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, Second edition, Oreilly & Associates, 2003.

[11] P. J. Denning, “The Working Set Model for Program Behavior,” Communications of the ACM, vol.11, no.5, May 1968.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation Techniques for Storage Hierarchies,” IBM Systems Journal, vol.9, no.2, pp.78-117, 1970.

[13] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” Proc. of 20th International Conference Very Large Data Bases, pp.439-450, 1994.

[14] P. E. O'Neil and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering,” Proc. of ACM SIGMOD Conference, May 1993.

[15] Y. Smaragdakis, S. Kaplan, and P. Wilson, “EELRU: Simple and Effective Adaptive Page Replacement,” Proc. of ACM SIGMETRICS Conference, 1999.

[16] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, “LRFU: A Spectrum of Policies that Subsumes the LRU and LFU Policies”, IEEE Transactions on Computers, vol. 50, no. 12, pp.1352-1361, Dec. 2001.

[17] N. Megiddo and D.S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache,” Proc. of USENIX Conference File and Storage Technologies (FAST), 2003.

[18] N. Young, “The k-server Dual and Loose Competitiveness for Paging,” Algorithmica, vol.11, no. 6, pp. 525-541, June 1994.

[19] J. Jeong and M. Dubois, “Optimal Replacements in Caches with Two Miss Costs,” Proc. of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, 1999.

[20] J. Jeong and M. Dubois, “Cost-sensitive Cache Replacement Algorithms,” Proc. of the Symposium on High-Performance Computer Architecture (HPCA), Jan. 2003.

[21] H. 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, Apr. 2005.