JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 8, No. 3, September 2014, pp.

Page Replacement for Write References in NAND Flash Based Virtual Memory Systems

Hyejeong Lee and Hyokyung Bahn*
Department of Computer Engineering, Ewha Womans University, Seoul, Korea
huizh@ewhain.net, bahn@ewha.ac.kr

Kang G. Shin
Department of Electrical Engineering and Computer Science, The University of Michigan, MI, USA
kgshin@umich.edu

Abstract
Contemporary embedded systems often use NAND flash memory instead of hard disks as their swap space of virtual memory. Since the read/write characteristics of NAND flash memory are very different from those of hard disks, an efficient page replacement algorithm is needed for this environment. Our analysis shows that temporal locality is dominant in virtual memory references but that is not the case for write references, when the read and write references are monitored separately. Based on this observation, we present a new page replacement algorithm that uses different strategies for read and write operations in predicting the re-reference likelihood of pages. For read operations, only temporal locality is used; but for write operations, both write frequency and temporal locality are used. The algorithm logically partitions the memory space into read and write areas to precisely keep track of their reference patterns, and then dynamically adjusts their size based on their reference patterns and I/O costs. Without requiring any external parameter to tune, the proposed algorithm outperforms CLOCK, CAR, and CFLRU by 20%–66%. It also supports optimized implementations for virtual memory systems.

Category: Embedded computing

Keywords: Memory; Secondary storage; Storage hierarchies; Swapping; Virtual memory; Flash memory

I. INTRODUCTION

NAND flash memory has become the most popular secondary storage media in modern embedded systems, such as smartphones, tablets, and portable media players (PMPs). As these embedded systems provide an increasing variety of functions, virtual memory support with swap space is becoming an important issue. Since the traditional swap space (i.e., the hard disk) of virtual memory system is now being replaced by NAND flash memory, an efficient virtual memory management technique is necessary for this emerging environment.

However, NAND flash memory is known to possess significantly different physical characteristics from hard disks. As a result, flash translation layers (FTLs) and flash-specific file systems have been extensively studied [1-8]. Unlike these studies, research on virtual memory systems for NAND flash memory is in its infancy [9-11].

Open Access http://dx.doi.org/10.5626/JCSE.2014.8.3.00 http://jcse.kiise.org

Received 0 A 2014; **Revised** 0 A 2014; **Accepted** 0 A 2014

*Corresponding Author

★ A subset of this paper was presented at the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009).

 pISSN: 1976-4677 eISSN: 2093-8020

In this paper, we analyze the page reference characteristics of a virtual memory system that uses NAND flash memory as its swap space, and present a new page replacement algorithm for this environment.

Page references in a virtual memory environment have a temporal locality property, and thus, the least recently used (LRU) and its approximated CLOCK algorithms have been widely used. However when replacing clean and dirty pages, they do not consider the different I/O costs of read and write operations in NAND flash memory. Dirty pages need to be swapped out or flushed to NAND flash memory before their eviction, and this incurs a write I/O that is about 3–10 times slower than a read I/O [10, 12, 13]. Note that a dirty page is one that has been modified during its residence in the memory, while a clean page is one that has not been changed. Thus, an efficient replacement algorithm needs to take account of these asymmetric operation costs. Furthermore, when read and write references are independently observed, the temporal locality of virtual memory references should be revisited.

In this paper, we analyze the characteristics of virtual memory read and write references separately in terms of their temporal locality. We have discovered an important phenomenon from this analysis; unlike read references that exhibit strong temporal locality, the temporal locality of write references is weak and irregular. Specifically, more recently written pages do not incur more writes in future for a certain range of page rankings. We call this phenomenon the *ranking inversion* of write temporal locality. Accordingly, temporal locality has limitations in predicting future references for write operations.

Based on this observation, we propose a new page replacement algorithm that uses different strategies for read and write operations in predicting the re-reference likelihood of pages. For read operations, temporal locality alone is considered; but for write operations, write frequency is used as well as temporal locality. With this idea, the new algorithm individually keeps track of the reference patterns of read and write operations, and accurately predicts the likelihood of re-referencing pages. It logically partitions the memory space into read and write areas based on the different I/O costs of operations. Then, it dynamically adjusts the size of each area according to the change of access patterns. In each area, the recency of page references is separately captured using a CLOCK list. Our experimental results with various virtual memory access traces show that the proposed algorithm, called *Clock for read and write* (CRAW), significantly improves the I/O performance of a virtual memory system. Specifically, it reduces I/O time by 20%–66%, compared to widely known algorithms, such as CLOCK, CAR, and CFLRU.

Moreover, in contrast to LRU, which needs to perform list manipulations or time-stamping on every memory reference, CRAW does not require either time-stamping or list manipulations unless page faults occur. This makes CRAW suitable for virtual memory environments that allow OS controls only upon page fault. Moreover, the parameters of CRAW are automatically tuned differentiating itself from the other approaches that require manual parameter tuning to deal with workload changes. The main contributions of this paper can be summarized as follows:

1) To capture the characteristics of write operations that are responsible for a large portion of I/O cost in flash memory, we separately analyze the temporal locality of memory accesses for read and write references.
2) In the case of read references, we have discovered that temporal locality is strong, and thus recency-based algorithms, such as CLOCK, are suitable for estimating the re-reference likelihood of read references.
3) In the case of write references, we have uncovered a prominent phenomenon that temporal locality is weak and irregular. Hence, estimating the re-reference likelihood of write references with temporal locality only is limited. We propose a new way to estimate future references by considering write frequency as well.
4) Considering the different I/O costs and reference characteristics of read and write operations, we separately allocate memory space for reads and writes, and dynamically change the allocated space according to the evolution of workloads.
5) We also show how the proposed algorithm can be easily deployed in various system environments without any modification of existing hardware architectures.

The rest of the paper is organized as follows. Section II discusses some related work on page references in virtual memory and reviews replacement algorithms for flash memory. In Section III, we capture page reference characteristics in virtual memory in terms of read and write references, and analyze the observation results. Section IV presents a new page replacement algorithm for virtual memory systems built on NAND flash memory. Then, Section V presents our experimental results obtained through trace-driven simulations to assess the effectiveness of the proposed algorithm, CRAW. Section VI describes some practical issues related to the deployment of CRAW in real system architectures. Finally, we conclude this paper in Section VII.

II. RELATED WORK

In this section, we briefly describe the characteristics of page references in virtual memory environments, and review existing page replacement algorithms for NAND flash memory.

A. Page References in Virtual Memory

Page references in a virtual memory environment have temporal locality in that a more recently referenced page is more likely to be referenced again soon. In terms of the hit ratio, the LRU replacement algorithm is known to be optimal for references that exhibit this property [14]. LRU aligns all the pages in the memory in the order of their most recent reference time and replaces the LRU page, whenever free page frames are needed. It is the most popular replacement algorithm in various caching environments including file system buffer cache, since it performs well, but has only a constant time and space overhead.

Nevertheless, LRU has a critical weakness in virtual memory environments. On every memory reference, LRU needs to move a page to the most recently used (MRU) position in the list. This involves some list manipulations, which cannot be handled by the paging unit hardware. Though LRU can also be implemented by hardware, this is not feasible in virtual memory systems as it should maintain the time-stamp of each page and update it upon every memory reference.

CLOCK was introduced as a one-bit approximation to LRU [15]. Instead of keeping pages in the order of reference time, CLOCK only monitors whether a page has recently been referenced or not. On each hit to a page, the paging unit hardware sets the reference bit of the page in the page table entry to 1. Then, pages are maintained in a circular list. Whenever free page frames are needed, CLOCK sequentially scans through the pages in the circular list starting from the current position, that is, next to the position of the last evicted page. This scan continues until a page with a reference bit of 0 is found, and that page is then replaced. For every page with reference bit of 1 in the course of the scan, CLOCK clears the reference bit to 0 without removing the page from the list.

The reference bit of each page indicates whether that page has recently been accessed or not; and a page that is not referenced until the clock-hand comes round to that page again is certain to be replaced. Even though CLOCK does not replace the LRU page, it replaces a page that has not been recently referenced so that temporal locality is exploited to some extent. In addition to this, since it does not require any list manipulation on memory hit, CLOCK is suitable for virtual memory environments.

Not recently used (NRU) is another version of page replacement algorithm that exploits the temporal locality of virtual memory environments. NRU works similarly to CLOCK using reference bits but it also uses modified bits to distinguish clean and dirty pages [16]. While reference bits are periodically cleared by the clock-hand to monitor the recency of page references, the modified bit of a page is set when the page becomes dirty; and is not cleared until it is evicted from memory. The modified bit indicates that the corresponding page should be written back to secondary storage before eviction. There are four different cases according to the state of the reference and modified bits.

NRU preferentially replaces pages whose reference bit is 0 to consider temporal locality. Of them, NRU evicts the pages with zero modified bit first because they are clean pages and can thus be evicted without incurring additional write I/O operations. Similarly, among pages with reference bit set to 1, NRU gives higher priorities to pages with a modified bit of 1. In summary, the page replacement order of NRU according to the state of (reference bit, modified bit) is (0, 0), (0, 1), (1, 0), and (1, 1).

B. Page Replacement for Flash Memory

Most operating systems, including Linux, are optimized under the assumption that secondary storage devices will be hard disk drives, which have almost identical costs for read and write operations. Page replacement algorithms therefore focus on maximizing the hit ratio by replacing the page least likely to be referenced again. In this process, the type of operation (read or write) that would be involved in that reference is not considered. Unlike hard disks, however, NAND flash memory has asymmetrical read and write costs. In NAND flash memory, servicing a write I/O request takes 3–10 times longer than servicing a read I/O request for the same I/O size as shown in Table 1 [12, 13].

In addition, NAND flash memory requires an erase operation before writing data on the same place again. Most systems, therefore, have a FTL, which hides the cost of erase operations by performing out-of-place-updates. As a result, traditional page replacement algorithms that aim to maximize the hit ratio do not perform well in the systems based on NAND flash memory because their performance metric is the hit ratio, although it should be the I/O time.

Clean-first LRU (CFLRU) [10] is a new page replacement algorithm that considers the hit ratio as well as the physical characteristics of NAND flash memory in which reading and writing have different I/O costs. CFLRU can accommodate the different eviction costs of a clean page, which can simply be discarded, and a dirty page, which should be written back to flash memory. CFLRU delays

Table 1. Read and write performance of NAND flash memory

	NAND flash memory		NAND-based SSD	
	SLC (2 kB page)	MLC (4 kB page)	Random (4 kB IOPS)	Sequential (bandwidth)
Read	25 μs	60 μs	35,000 IOPS	250 MB/s
Write	200 μs	800 μs	3,300 IOPS	70 MB/s
Read : Write	1 : 8	1 : 13.3	1 : 10.6	1 : 3.6

SSD: solid state disk, SLC: single-level cell, MLC: multi-level cell.

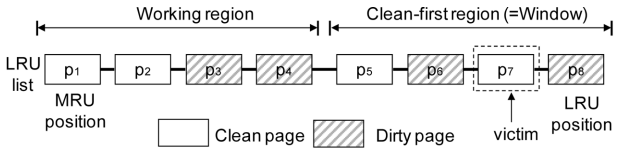

Fig. 1. An example of page replacement in clean-first LRU (CFLRU). LRU: least recently used, MRU: most recently used.

the eviction of dirty pages to reduce the number of writes to NAND flash memory, unless this will do too much harm to the hit ratio. Though NRU also delays the replacement of dirty pages, it uses the modified bit only as a tiebreaker among pages with the reference bit offset to 0.

CFLRU manages pages using the LRU list. CFLRU divides this list into a working region and a clean-first region as shown in Fig. 1. The working region contains recently referenced pages that are replaced according to the LRU policy. The evicted page from the working region is inserted into the clean-first region, which contains older pages whose last reference was made a relatively long time ago. Dirty pages in the clean-first region are preserved in the memory as long as a clean page is available for eviction. CFLRU starts by searching the clean-first region for a candidate for replacement. The length of the clean-first region is defined as a fraction of the total memory size, called the *window size*. Fig. 1 shows an example of page replacement in CFLRU. P_8 is the LRU page; but when a free page frame is necessary, CFLRU replaces P_7, which is the LRU clean page. Since dirty pages can only be evicted if no clean page exists in the window, pages are evicted in the order of P_7, P_5, P_8, and finally P_6.

CFLRU was the first to adapt LRU for NAND flash memory-based systems. However, its favor on dirty pages varies greatly across the boundary between the working region and the clean-first region. Moreover, the window size should be tuned as the workload changes. As a consequence, CFLRU does not cope well with changes in workload characteristics, such as the ratio of read to write accesses.

LRU with write sequence reordering (LRU-WSR) is another replacement algorithm that favors dirty pages [17]. Basically, LRU-WSR also manages pages using the LRU list. Instead of setting the clean-first region, LRU-WSR gives one more chance to a dirty page, when it reaches the LRU position in the list. Specifically, LRU-WSR keeps a cold flag for each page in the LRU list. When a page is referenced, LRU-WSR moves the page to the MRU position of the list. Additionally, if it is a dirty page, LRU-WSR clears the cold flag of that page. When replacement is needed, LRU-WSR checks the page in the LRU position. If the page is clean, it is replaced. Otherwise, the cold flag is checked. If it is set to 1, LRU-WSR replaces the page. If the cold flag of the page is 0, LRU-WSR sets the cold flag to 1, moves the page to the MRU position to give one more chance, and checks another page at the LRU position. In this way, LRU-WSR considers asymmetric operation costs of reads and writes in the flash memory, but it still does not consider their exact costs in the algorithm design.

There are some categories of replacement algorithms that exploit the device-specific information of NAND flash memory. Since most of this information cannot be delivered to virtual memory and/or file systems in current system interfaces, the algorithms are usually targeted to device-specific buffer managers, or some specific systems. Flash-aware buffer management (FAB) is proposed as a replacement algorithm of DRAM buffer in flash-based PMP systems [18]. PMP systems commonly have long sequential accesses for media data and some short accesses for metadata at the same time. One problem with this situation is that short write accesses cannot be buffered for a long time because they are pushed away by a large amount of sequential data. This eventually incurs frequent random write I/Os leading to degraded I/O performances due to full merge operations in log-block FTLs [5, 19]. To cope with this problem, FAB manages buffered data from the same NAND flash block as a group and replaces them together. When free buffers are needed, FAB evicts a NAND block group with the largest number of buffers. If more than one group have the same largest number of buffers, the LRU order is used as a tiebreaker.

Block padding least recently used (BPLRU) is a write buffer management algorithm to improve the random write performance of flash storage in desktop environments [20]. BPLRU manages an LRU list for RAM buffers. Similar to FAB, BPLRU groups buffers from the same NAND flash block and replaces them together. When a buffer is accessed by a write operation, buffers in the same group are moved together to the MRU position of the list. BPLRU selects buffers in the LRU position as a victim and flushes all data in the group. This block-level flushing reduces the total merge cost of NAND flash memory in log-block FTLs. BPLRU also uses two heuristics, called *page padding* and *LRU compensation*. Page padding makes a partially buffered NAND block into a fully buffered one by reading the rest of NAND pages from the flash, just before evicting the block. LRU compensation is a heuristic that evicts a fully buffered NAND block first.

Cold and largest cluster (CLC) is another write buffer replacement algorithm for NAND flash memory [21]. Unlike FAB and BPLRU, CLC uses byte-addressable non-volatile memory as its write buffer. Similar to FAB and BPLRU, CLC manages pages from the same NAND flash blocks together. When replacement is needed, CLC selects a NAND block group with the largest number of pages among groups that have not recently been referenced.

III. PAGE REFERENCES IN VIRTUAL MEMORY

In this section, we analyze and capture page references in virtual memory systems specially focusing on the *temporal locality* of read and write operations. For write operations, we also analyze the *write frequency* as well as *temporal locality* to more precisely characterize the page reference behavior. We capture the virtual memory access traces from four different applications used on Linux X Windows, namely, the xmms mp3 player, the gqview image viewer, the gedit word processor, and the freecell game. The characteristics of these traces will be explained later in Section V. We first analyze the temporal locality of total references in the traces, and then classify them into read and write references to more precisely examine the re-reference likelihood of each operation.

A. Temporal Locality

As shown in Fig. 2, virtual memory accesses exhibit strong temporal locality. In the figure, the *x*-axis represents the page ranking in the LRU list (i.e., the LRU stack distance). For example, the ranking 1 in the *x*-axis refers to the page at the most recently referenced position in the LRU list. Increased ranking of the *x*-axis increases means that the pages were referenced a relatively longer time ago. The *y*-axis represents the number of references that occur for the page ranking in the *x*-axis.

As shown in Fig. 2, the shape of the curve can be well modeled by a monotonic decreasing function implying that a more recently referenced page is more likely to be referenced again. For this reference pattern, the LRU algorithm is known to perform well [14].

Fig. 2 shows the temporal locality of total page references including both read and write references. Figs. 3 and 4 separately show the temporal locality of read and write references. For example, the *x*-axis in Fig. 3 represents the recency ranking of read references and the *y*-axis represents the number of read references that occur for the given ranking. As shown in Fig. 3, read references exhibit strong temporal locality. Unlike the plots in Fig. 2(a) and (b) that contain some projecting points, the plots in Fig. 3(a) and (b) are more fluent. This means that the temporal locality of read references is stronger than that of total references including both read and write references.

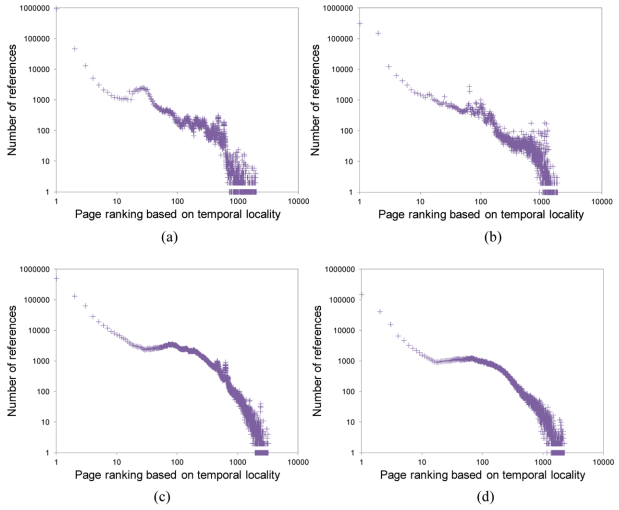

Fig. 2. Reference counts versus temporal locality ranking for total references including reads and writes. (a) xmms, (b) gqview, (c) gedit, and (d) freecell.

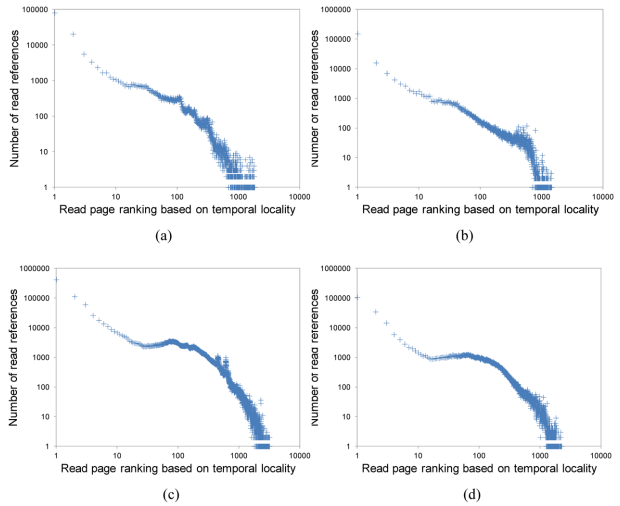

Fig. 3. Number of read references occurred versus temporal locality ranking of read references. (a) xmms, (b) gqview, (c) gedit, and (d) freecell.

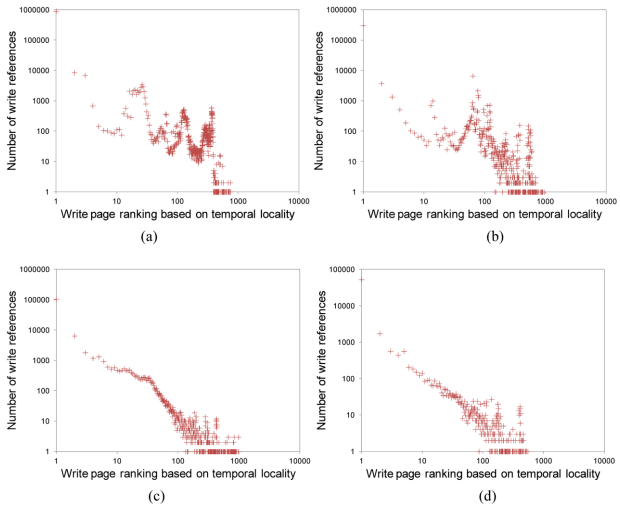

Fig. 4. Number of write references occurred versus temporal locality ranking of write references. (a) xmms, (b) gqview, (c) gedit, and (d) freecell.

Now, let us examine the write references. As shown in Fig. 4, the temporal locality of write references is rather irregular. Specifically, Fig. 4 shows the ranking inversion of temporal locality, i.e., a more recently written page shows a smaller fraction of re-writes in some ranges of ranking. We can clearly observe this phenomenon from Fig. 4(a) and (b), which contain a relatively large number of write references.

Based on this observation, we can conclude that temporal locality alone is not sufficient to estimate the re-reference likelihood of write references in virtual memory. We cannot pinpoint the exact reason for this phenomenon, but we conjecture that it is due to the write-back operation of the CPU cache memory. Since a certain portion of memory references are absorbed by the cache memory, page references observed at the virtual memory layer contain only the references that are cache-missed. In the case of read references, cache-missed requests directly propagate to the virtual memory layer, thus not much affecting temporal locality, although it becomes rather weak. However, in the case of write references, cache-missed requests do not propagate directly to virtual memory, but are just written to the cache memory. Then, the write references are delivered to virtual memory only after the data are evicted from the cache memory. This implies that the time a write request arrives is asynchronous with the time that the request is delivered to main memory. This is the reason why temporal locality of write references is considerably dispersed.

B. Frequency of Write References

In Section III-A, we observed that the temporal locality of write references in virtual memory is greatly dispersed. Here, we analyze the effect of write frequency on the re-reference likelihood of write references. We can consider two different types of frequency. The first is the total write frequency, which counts the total number of writes appearing in the trace; and the second is the so-far-write-frequency, which counts the number of writes that have occurred to the current point. We use the latter in order to observe the impact of frequency on estimating a page's re-reference likelihood each time in comparison with temporal locality. To do this, we maintain the ranking of pages according to their past write counts, and examine the number of write operations that occur again for each ranking.

In Fig. 5, the *x*-axis represents the ranking of pages based on their past write counts. The *y*-axis represents the number of writes that occur on that ranking. To construct the curve, we maintain the page ranking each time, and as a page in a certain ranking is written again, we increase the value of *y*-axis for that ranking by one possibly resulting in a reordering of the page rankings.

As shown in Fig. 5, most write references that are made are in the range of top ranking. This means that a page referenced frequently in the past is likely to be referenced again in the future. Unlike the temporal locality of write references, the frequency of write references does not show the ranking inversion problem. It also exhibits larger reference counts than temporal locality for a certain range of top ranking.

In summary, the re-reference likelihood of read references can be well modeled by temporal locality. For write references, however, using the write frequency as well as temporal locality will be more effective. To more accurately predict future write references, a page replacement algorithm should consider the write frequency as well as the recency of the references.

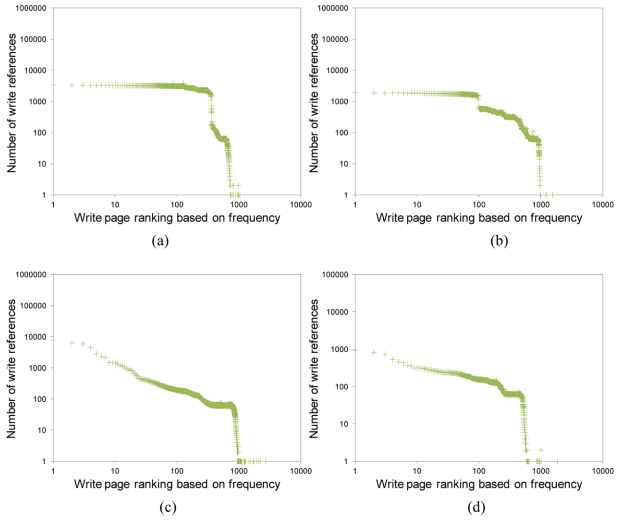

Fig. 5. Number of write references occurred versus frequency ranking of write references. (a) xmms, (b) gqview, (c) gedit, and (d) freecell.

IV. A NEW PAGE REPLACEMENT ALGORITHM

In this section, we present a new page replacement algorithm for virtual memory systems, called CRAW, which uses NAND flash memory as its swap device. CRAW separately allocates memory areas for read and write operations so as to minimize the total I/O costs. It does this by finding the contribution of each area and dynamically adjusting their size.

For each area, replacement is efficiently performed, similar to the implementation of the CLOCK algorithm. To select a victim page in each area, CRAW exploits the read and write characteristics of virtual memory explained in Section III. That is, for read references, temporal locality is exploited, and for write references, both temporal locality and write frequency are used to predict the re-reference likelihood of pages. All the pages in the write area are dirty pages, which need 3–10 times higher cost in terms of time to evict than clean pages. CRAW gives

higher priority to write pages that incur relatively high costs; but it also preserves read pages if they are frequently referenced, and thus their contribution to improving I/O performance is significant.

A. Adjusting the Size of Each Area

CRAW employs ghost areas to evaluate and adjust the size of the read and write areas as shown in Fig. 6. Ghost areas only maintain the metadata of recently evicted pages without their actual data. By observing references to a page in the ghost areas, CRAW predicts the effect that extending each area would have on performance. If there are frequent hits on pages in the ghost read area, CRAW extends the read area to reduce the number of page faults. The write area can be extended in the same way. In addition to the hits to pages in the ghost areas, the different cost of a read and a write is also considered in adjusting the size of each area.

The size of each ghost area is adjusted as the size of corresponding area changes such that the total number of pages in these two areas is equal to the total number of page frames, which is referred to as *S*. For example, after extending the read area to accommodate one more page, CRAW shrinks the ghost read area by one. This is because the hit ratio for the whole memory can be predicted, if the sum of the allocated pages and the ghost pages is equal to *S*. As explained in Fig. 6, *S* ghost pages are sufficient for both read and write areas because *S* pages are actually allocated. Maintaining this number of ghost pages has very low overhead because a ghost page only contains 20 bytes of information including pointers and a page identifier, whereas each of the actual pages contains 4 kB of data [22-26].

Fig. 6 briefly shows the read area R, the write area W, and their ghost areas. In practice, however, R and W may share pages. For example, a page that has been recently read, as well as written, is kept in both R and W. In this case, the page data is in one page frame, and the page descriptor is linked to both R and W using different link pointers in order to independently and accurately manage the areas. Consequently, |R|+|W| can be larger than *S*, and the number of ghost pages can be equal to or less than *S*. Since CRAW allows pages to be simultaneously linked to multiple areas, a page can be evicted from physical memory only if it is not linked to any area.

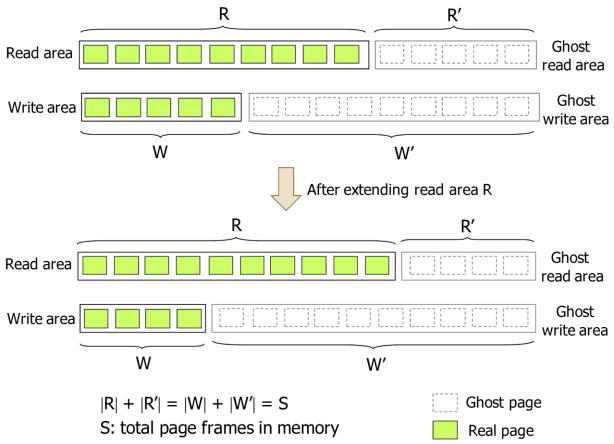

Fig. 6. Adjusting the sizes of the read and write areas by using ghost areas.

B. Details of the Algorithm

We now describe how the CRAW algorithm works in detail. Fig. 7 depicts the basic structure of CRAW. Pages in memory are managed by read area R and write area W. The metadata of evicted pages from these areas are kept in ghost read area R′ and ghost write area W′, respectively.

Page replacement for each area is independently managed by using an efficient design similar to the CLOCK algorithm. When CRAW searches for a victim to evict from the read area, it checks the read bit of the page to which the clock-hand points as CLOCK also does. If the read bit is 1, it is cleared; otherwise, the page is deleted from the area. The clock-hand scans clockwise through the pages until it finds a page with a zero read bit. When CRAW searches for a victim in the write area, the write bit instead of the read bit is investigated. In fact, the read and write bits have similar meanings to the reference and modified bits that are set by the paging unit hardware during every memory access. The metadata for pages deleted from R and W are inserted into ghost areas R′ and W′, respectively. When a replacement is needed in a ghost area, the least recently inserted page is deleted. Note that this is identical to the FIFO order in the ghost areas. As shown in Fig. 7, a newly inserted page is linked to the MRU location in the ghost area, and the oldest page is evicted from the LRU location.

Since write frequency is also important to predict the re-reference likelihood of write references, CRAW manages the internal structure of write area W by two partitioned sub-areas, namely write temporal locality area W1, and write frequency area W2. Ghost areas for W1 and W2 are also separately managed. Similar to R and W,

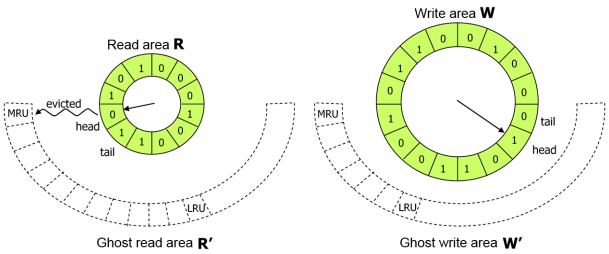

Fig. 7. Basic structure of Clock for read and write (CRAW). LRU: least recently used, MRU: most recently used.

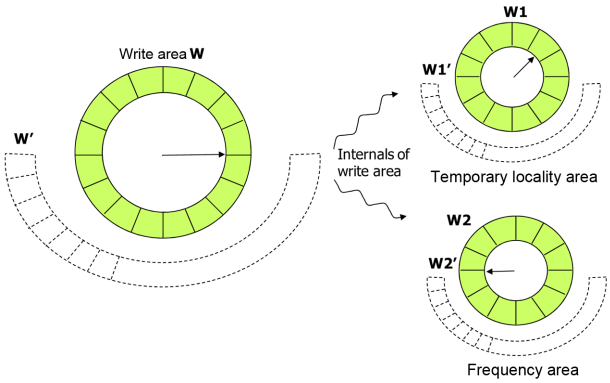

Fig. 8. The internal structure of the write area considering both temporal locality and write frequency.

the sizes of W1 and W2 are adjusted by page hits in ghost areas W1′ and W2′. W1 maintains pages that are written once while their metadata are resident in memory, and W2 maintains pages that are written more than once.

A page may not be directly included in the write area even after a write reference to that page happens. This is the same for read references. For example, if a page is not in write area W1 or W2 but exists in read area R, no page fault occurs upon a write reference, and thus list manipulation by kernel is not possible. In this case, the write bit of this page is just set by the paging unit hardware, and the page still remains only in the read area. The page is finally included in the write area when the write bit is found during the scanning of the read area to find a victim.

Now we will give further details of the CRAW algorithm following the pseudocode in Fig. 9. If the CPU references a page that is already in memory, CRAW does nothing except for bit setting. The read bit of the page is set on a read reference and the write bit of the page is set on a write reference.

When a page fault occurs because a referenced page is not in memory, CRAW first checks if there is a free page frame. If not, CRAW invokes the `reclaim`() function to get a free page frame by evicting a page from memory and then stores the requested page in the frame after retrieving it from secondary storage (we provide the description of CRAW from a theoretical aspect here. Actually, commodity operating systems do not invoke `reclaim`() function on demand but reserve a certain number of free page frames in advance). Then, CRAW inserts the page in the area corresponding to the access type and adjusts the size of their areas, if necessary.

If a page fault results from a read access, CRAW inserts the page at the tail position of the read area. If the history of the page exists in the ghost read area R′, it is deleted from R′, and if necessary, the size of R is increased. The size of R′ is then reduced to preserve the balance between R and R′.

If a page fault results from a write access, there are three different cases. First, if the history of the page exists in W1′, CRAW deletes it and inserts the page into the tail position of W2. In this case, a hit occurs in ghost write area W1′ and thus the size of W1 is increased, and if necessary, the sizes of other areas including ghost areas are adjusted. Second, if the history of the page exists in W2′, CRAW deletes it and inserts the page to the tail position of W2. In this case, a hit occurs in ghost write area W2′ thus the size of W2 is increased, and if necessary, the sizes of other areas including ghost areas are adjusted. Third, if the page history does not exist in any of ghost write areas, it is inserted to the tail position of W1.

Now, we describe the reclamation procedure used to obtain a free page frame. The `reclaim`() function first selects the area from which to evict a page. Since CRAW maintains and adjusts the desired size for each area according to the hits from ghost areas, it basically selects an area containing more pages than its desired size as the victim area.

In practice, however, there may be more than one area that satisfies this condition, because CRAW allows a page to be shared by multiple areas. In this case, CRAW selects the area that has the largest ratio of current size to the desired size as the victim area. For example, let us assume that the desired sizes of R, W1, and W2 are 3, 4, and 5, and the current sizes of them are 4, 5, and 5, respectively. In this example, both R and W1 have more pages than their desired sizes. Since R has a larger ratio of current to desired size, R is selected as the victim area in this case.

As explained earlier, when `reclaim`() chooses to evict a page from the read area, it first checks the read bit of the page to which the clock-hand points as CLOCK does. If the read bit is 1, `reclaim`() clears the bit and moves to the next page. Otherwise, `reclaim`() deletes the page from the area and returns. In this process, if a page is found with its write bit set in the read area, CRAW clears the write bit and links that page to the tail of write area W1, if the page is not already in the write area. If `reclaim`() selects W1 or W2 to evict a page, the write bit is checked in the same way. In this process, if a page is found with its read bit set to 1, CRAW clears the read bit, and links that page to the tail of read area R, if the page is not in R. This process is simple and fast because it only requires a bit and a list pointer to be checked, and some list manipulations are performed only when needed. Fig. 10 depicts the conceptual flow of the CRAW algorithm.

V. PERFORMANCE EVALUATION

We now present the performance evaluation results to assess the effectiveness of the CRAW algorithm. A trace-driven simulation is performed to manage the replacement algorithm of a virtual memory system with accurate

```
S is memory size and c is the cost ratio of a write to a read;
S_R, S_W1, S_W2 are desired sizes of R, W1, W2, respectively;
Initially S_R = S/c, S_W1 = (S–S_R)/2, S_W2 = (S–S_R)/2;

procedure CRAW(page p, operation op)
  if(p is in memory) then
    if(op is read)  then read_bit(p) = 1;
    else  write_bit(p) = 1;
    end if
  else /* page fault */
    while(no free page in memory) do
      RECLAIM();
    end while
    MEMORY_ADD(p, op);
    ADJUST_GHOST_SIZE();
  end if
end procedure

procedure MEMORY_ADD(page p, operation op)
  if(op is read) then
    if(p∈R′) then
      remove p from R′;
      if(pages in R′  has accessed c times) then
        S_R = min(S_R + 1, S); /* increase S_R */
        S_W1 = max(S_W1 – 0.5, 0); /* decrease S_W1 */
        S_W2 = max(S_W2 – 0.5, 0); /* decrease S_W2 */
      end if
    end if
    insert p at the tail of R;
    read_bit(p) = 0;
  else  /* op should be write */
    if(p∈W1′) then
      remove p from W1′ and insert p at the tail of W2;
      S_W1 = min(S_W1 + 1, S); /* increase S_W1 */
      S_R = max(S_R – 1, 0); /* decrease S_R */
    else if(p∈W2′) then
      remove p from W2′ and insert p at the tail of W2;
      S_W2 = min(S_W2 + 1, S);  /* increase S_W2 */
      S_R = max(S_R – 1, 0);  /* decrease S_R */
    else insert p at the tail of W1;
    end if
    write_bit(p) = 0;
  end if
end procedure

procedure ADJUST_GHOST_SIZE ()
  while(|R|+|R′| > S  and  |R′| > 0) do
    remove the LRU page from R′;
  end while
  while(|W1|+|W1′|+|W2|+|W2′|>S and |W1′|+|W2′|>0) do
    remove the LRU page from W1′ or W2′ in turns;
  end while
end procedure

procedure RECLAIM()
  r = |R|/S_R;  w1 = |W1|/S_W1;  w2 = |W2|/S_W2;
  /* ratio of current size to desired size */
  if(r ≥ w1  and  r ≥ w2) then  /* reclaim from R */
    while(1)  do
      p = clock-hand of R;
      clock-hand of R points to the next page;
      if(write_bit(p) is 1  and  p∉W1 ∪ W2) then
        insert p at the tail of W1;  write_bit(p) = 0;
      end if
      if(read_bit(p) is 1) then
        read_bit(p) = 0;  move p to the tail of R;
      else  /* read_bit(p) should be 0 */
        reclaim p from R and insert p at MRU position in R′;
        if(p∉W1 ∪ W2) then return(p); /* found free page */
        else return(NULL);
        end if
      end if
    end while
  else if(w1 ≥ r  and  w1 ≥ w2)  then  /* reclaim from W1 */
    while(1) do
      p = clock-hand of W1;
      clock-hand of W1 points to the next page;
      if(read_bit(p) is 1  and  p∉R) then
        insert p at the tail of R;  read_bit(p) = 0;
      end if
      if(write_bit(p) is 1) then
        write_bit(p) = 0;  move p to the tail of W2;
      else  /*  write_bit(p) should be 0 */
        reclaim p from W1 and insert to MRU position in W1′;
        if(p∉R) then return(p);  /* found free page */
        else return(NULL);
        end if
      end if
    end while
  else if(w2 ≥ r and w2 ≥ w1) then /* reclaim from W2 */
    while(1) do
      p = clock-hand of W2;
      clock-hand of W2 points to the next page;
      if(read_bit(p) is 1 and p∉R) then
        insert p at the tail of R;  read_bit(p) = 0;
        end if
      if(write_bit(p) is 1) then
        write_bit(p) = 0;  move p to the tail of W2;
      else  /*  write_bit(p) should be 0 */
        reclaim p from W2 and insert to MRU position of W2′;
        if(p∉R) then return(p);  /* found free page */
        else return(NULL);
        end if
      end if
    end while
  end if
end procedure
```

Fig. 9. Pseudocode of Clock for read and write (CRAW).

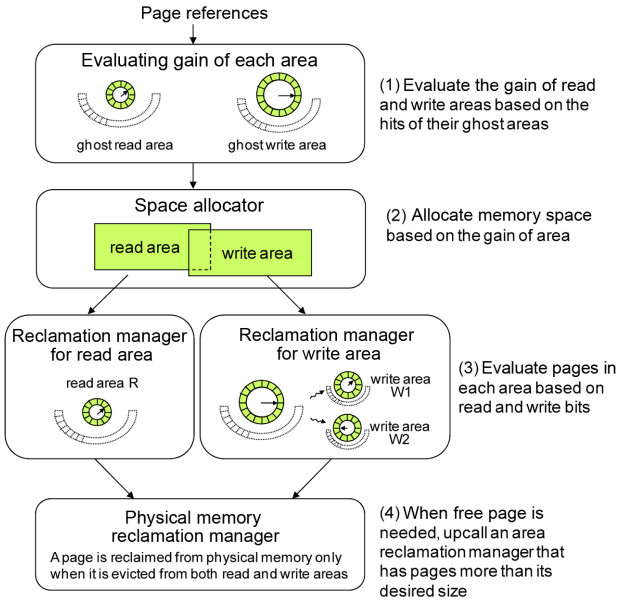

Fig. 10. A conceptual flow of Clock for read and write (CRAW).

I/O timing of NAND flash memory, and software overheads of replacement algorithms. The size of a virtual memory page is set to 4 kB, which is common to most operating systems including Linux. Secondary storage is assumed to consist of a large block SLC NAND flash memory, where a block contains 64 pages, and a page has 2 kB of data. The I/O time of a read and a write operation for each flash page is set to 25 μs and 200 μs, respectively, as listed in Table 1.

A. Experimental Setup

The traces are acquired by a modified version of the Cachegrind tool from the Valgrind 3.2.3 toolset [27, 28]. Fig. 11 shows the form of each request in the trace. Memory accesses in the traces can be classified into instruction reads, data reads, and data writes.

We capture the virtual memory access traces from six different applications used on Linux X Windows, namely, the xmms mp3 player, the gqview image viewer, the gedit word processor, the freecell game, the kghostview PDF file viewer, and the tar_gzip archiving and compression utilities. We filter out memory references that are accessed directly from the CPU cache memory, and also reflect the write-back property of the cache memory. The characteristics of these traces are described in Table 2.

Reference type	Virtual address	Access size (byte)
readi	0x04000BE0	2
write	0xBEFFFACC	4
readi	0x04000C30	1
write	0xBEFFFABC	4
readd	0x0401582C	4
:	:	:

Fig. 11. Form of each request in the trace.

The xmms trace shows a lot of write operations even though it is a multimedia playing program. Multimedia players seldom make write I/O requests upon the data file, but in the virtual memory, it decodes the data to a playable data stream and writes the decoded data, which is usually much larger than the encoded data to another page frame.

We compare the performance of CRAW with CLOCK, CAR (Clock with adaptive replacement) [22], and CFCLOCK [10]. CFCLOCK is a modified version of CFLRU to work efficiently in virtual memory environments. On hit to a page, CFCLOCK just sets the reference bit or modified bit and does not perform any list manipulations. Similar to CFLRU, CFCLOCK has a window size parameter that specifies how many pages it should first search for clean pages starting from the current clock-hand position. If it succeeds in finding a clean page with a reference bit set to 0 in the window, CFCLOCK replaces it. Otherwise, it scans through the pages in the window again and replaces a dirty page whose reference bit is not set. If there is no such page, then CFCLOCK searches for a page with a reference bit set to 0 starting from the next position. Whenever a free page frame is needed, CFCLOCK first scans through the window size of pages. In our experiments, the window size is set to

Table 2. Memory usage and reference count for each workload

Workload	Memory footprint (kB)	Ratio of operations (read : write)	Total access count
xmms	8,050	1 : 5.13	1,168,939
gqview	7,430	1 : 1.30	610,685
gedit	14,460	12.05 : 1	1,733,763
freecell	10,080	7.16 :1	490,175
kghostview	17,390	13.93 : 1	1,546,135
tar_gzip	752	5.23 : 1	4,535

one third of the total number of page frames according to the window size of the swap system used in the original CFLRU simulations [10].

B. Experimental Results

The performance of page replacement algorithms is measured by the total I/O time for a given workload. Fig. 12 shows the total I/O time of the four algorithms as a function of the memory size ranging from 1% to 100% of maximum memory usage of the traces. The 100% memory size means the unrealistic condition that a complete memory footprint can be loaded at the same time, and thus page replacement is not needed. In this environment, all algorithms perform the same. For each workload, the graphs show the total I/O time of each algorithm normalized to the CLOCK algorithm.

CRAW outperforms the other algorithms for a wide range of memory size and a variety of workloads. In comparison with CLOCK, CRAW reduced the total I/O time by an average of 23.9% and up to 66.5%. The performance improvements of CRAW over CAR and CFCLOCK are in the range of 25%–66% and 16%–58%, respectively.

Although CFCLOCK reduces the number of expensive write I/O operations by preserving dirty pages as much as possible, it falls behind CLOCK and CAR in dealing with some read-intensive workloads, such as freecell and gedit, for small memory sizes. Note that these two traces are read-intensive and their locality is also strong. Since CFCLOCK provides too much memory space to dirty pages, it fails to preserve a large amount of clean pages that only incur read operations. This shows that CFCLOCK is unable to adapt to workload changes in some read-intensive jobs. On the other hand, CRAW dynamically adapts to the changes of workload pattern and memory capacity resulting in consistently good performances.

Figs. 13 and 14 show the number of read and write I/Os performed on flash memory for each workload. This figure shows how CRAW could reduce total I/O time by compromising the cost of read and write operations for given workload and memory space.

In the case of xmms and gqview traces, there are more write references than read references. In these environments, CRAW preserves dirty pages in the memory as much as possible by enlarging write areas, which eventually reduces the number of write I/Os and total I/O time. In the case of gedit and freecell traces, however, read ref-

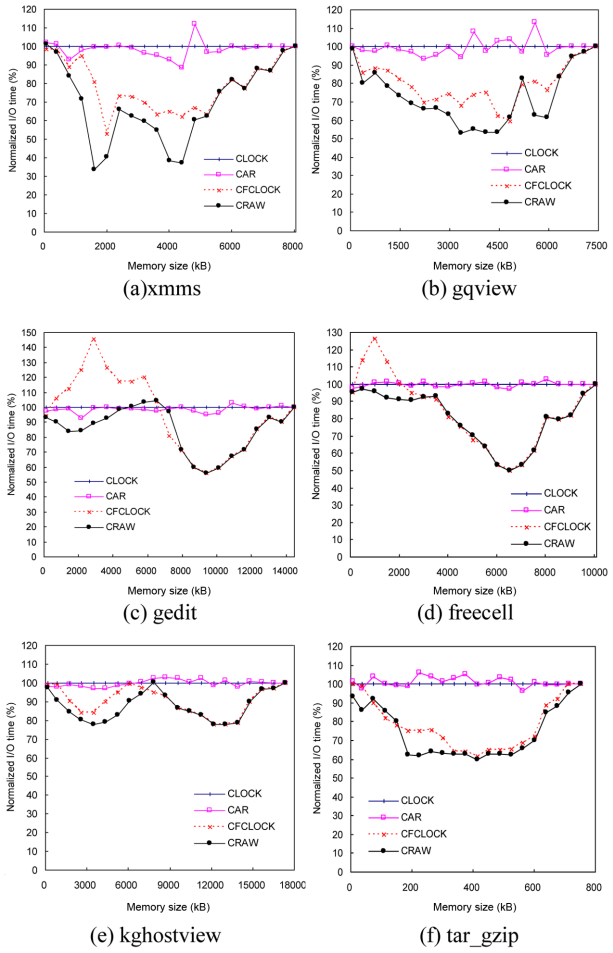

Fig. 12. Total I/O time for CLOCK, CAR, CFCLOCK, and CRAW as a function of the memory size. (a) xmms, (b) gqview, (c) gedit, (d) freecell, (e) kghostview, and (f) tar_gzip. CAR: Clock with adaptive replacement, CFCLOCK: , CRAW: Clock for read and write.

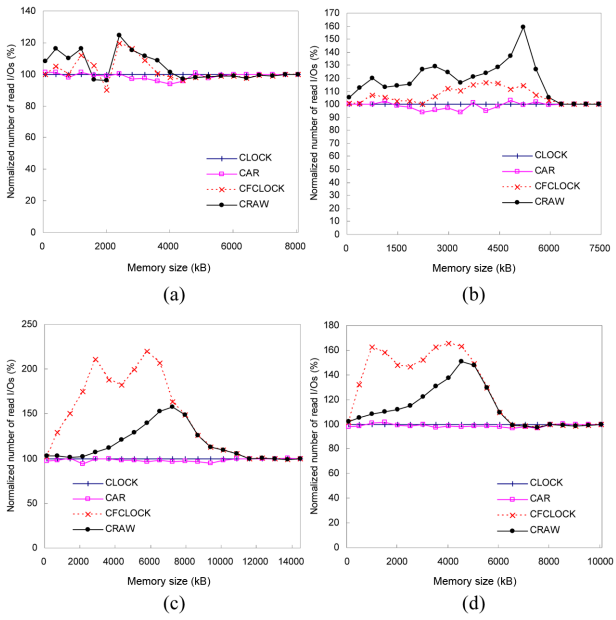

Fig. 13. Number of read I/Os made by CLOCK, CAR, CFCLOCK, and CRAW as a function of the memory size. (a) xmms, (b) gqview, (c) gedit, and (d) freecell. CAR: Clock with adaptive replacement, CFCLOCK: , CRAW: Clock for read and write.

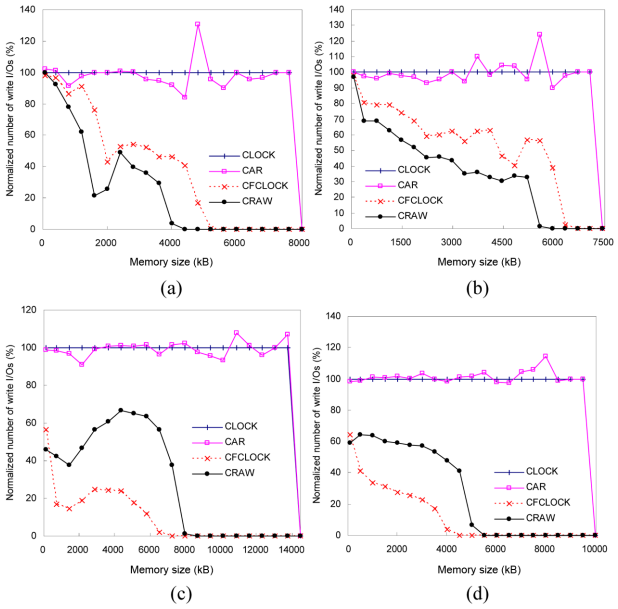

Fig. 14. Number of write I/Os made by CLOCK, CAR, CFCLOCK, and CRAW as a function of the memory size. (a) xmms, (b) gqview, (c) gedit, and (d) freecell. CAR: Clock with adaptive replacement, CFCLOCK: , CRAW: Clock for read and write.

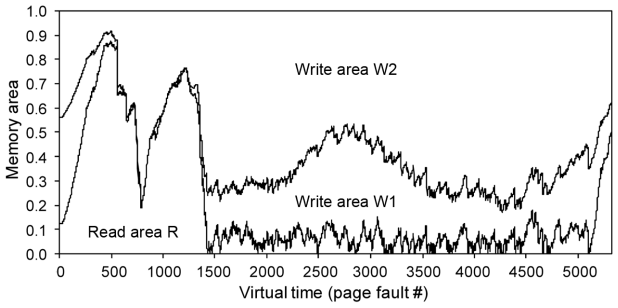

Fig. 15. The size of each area as time progresses.

erences are dominant. As shown in Figs. 13(c)–(d) and 14(c)–(d), CRAW focuses on reducing the number of expensive write I/Os, but it also keeps the number of read I/Os to a certain limited range. This leads to improved performance in terms of the total I/O time. CFCLOCK dramatically reduces write I/Os, but it performs even worse than CLOCK and CAR, when the memory size is relatively small. Observe the amount of read I/Os of CFCLOCK in Fig. 13(c) that is almost 200% of CLOCK.

Fig. 15 shows how CRAW changes the size of each area, as time progresses. The figure plots the desired sizes of the read area and the two write areas at each time a page fault occurs when the xmms trace is used. As shown in the figure, we can notice that the memory space is dynamically allocated according to the change of reference patterns.

C. Overhead of CRAW

Traditional real-time embedded systems simultaneously load the whole address space of a process into physical memory, instead of using virtual memory, thereby providing deadline-guaranteed services. However, as contemporary embedded systems provide multitasking, virtual memory is being supported and page faults inevitably occur. This paper focuses on such systems, and thus its goal is not pursuing the deadline-guaranteed service but reducing the overhead of page faults. Compared with the page fault handling process that accompanies slow storage accesses, the additional software overhead of CRAW is quite small. Actually, the additional software overhead of CRAW is already reflected in our experimental results shown in Fig. 12, which confirms that the reduced storage I/O overhead by CRAW has greatly influenced the overall performances.

The time complexity of CRAW is identical to that of the original CLOCK algorithm, in which the only non-constant part is involved in the clock-hand scanning process to find the replacement victim. In this process, the worst case time complexity is O(n), where n is the number of page frames. However, in practical situations, the scanning requires only a few movements of the clock-hand, implying that in practical terms, it has constant time complexity. Actually, worst case analysis like time complexity analysis does not consider the practical situations of real system environments, but just uses unrealistic conditions for worst cases to the algorithm. For example, LRU has the time complexity of O(1), even though its overhead is much larger than that of CLOCK in real systems. We believe that our experimental results cover a wide variety of cases, including the worst cases, as well as the common cases of real system environments, which show that the overhead of CRAW is sufficiently small.

VI. REALIZATION IN REAL SYSTEM ARCHITECTURES

In this section, we describe how the CRAW algorithm can easily be deployed in existing system architectures. The CRAW algorithm can be realized when the read and write bits are supported in the paging unit hardware. This is done by simple modification of bit settings in the existing architectures. Specifically, current paging unit hardware sets the reference bit to 1 when a read or a write reference occurs, and the modified bit to 1 when a write reference occurs. Instead of this setting, a new version of the paging unit hardware should set the reference bit only for a read reference. This simple modification allows the implementation of the original CRAW algorithm in real systems. However, we show in this section that CRAW can also be used in the existing system architectures with-

out hardware modifications by some approximated implementation or software support.

A. Implementation with Reference and Modified Bits

In the system architectures that support reference and modified bits, it may not be feasible to manipulate the read and write bits. Since the reference bit is set by either a read or a write operation extracting read information alone from the reference bit is a challenging problem. In such a case, we need to consider an approximated implementation of the read area of CRAW by using the reference bit.

We tried to use the total reference area (including both read and write references) and the write area instead of the read area and the write area. In this case, if a page is written, it is included into both the total reference area and the write area. To manage the two areas, reference and modified bits can be used instead of read and write bits. Since the original CRAW also allows duplication of a page in both areas, this modified version can work reasonably well. This is also consistent with the analysis shown in Section III, in which the temporal locality of read references is very similar to that of the total references. Fig. 16 depicts this approximated implementation of the CRAW algorithm, called CRAW-A.

To quantify the effect of CRAW-A on the performance of virtual memory systems, we compare the previous results in Fig. 12 with CRAW-A. From these experiments, we found that the performance of CRAW-A is almost identical to the original CRAW implementation. As shown in Fig. 17, for most cases, the performances of CRAW and CRAW-A are so similar that they are almost impossible to distinguish. Furthermore, in some cases, CRAW-A performs better than the original CRAW. From this result, we can conclude that CRAW can be effectively implemented as CRAW-A in existing system architectures without any modification of the paging unit hardware.

B. Implementation without Additional Bits

Some embedded system architectures do not provide reference and/or modified bits for the paging unit hardware. Even though such bits are not provided by hardware, the system still needs to keep track of the reference and modified bits to determine the replacement victim and flushing target. Thus, in this architecture, setting the reference and modified bits is usually handled not by hardware but by kernel in the way of intentional page faults.

In the case of the ARM architecture, when a page is first created, it is marked as read-only [29]. The first write to such a page (a clean page) will cause a permission fault, and the kernel data abort handler will be called. The memory management code will mark the page as dirty, if the page should indeed be writable. The page table entry is modified to make it allow both reads and writes, and the abort handler then returns to retry the faulting access in the application. A similar technique is used to emulate the reference bit, which shows when a page has been accessed. Read and write accesses to the page generate an abort. The reference bit is then set

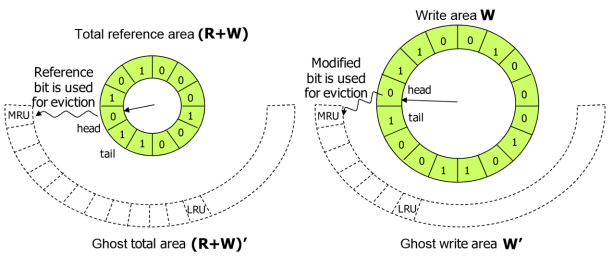

Fig. 16. CRAW-A: an approximated implementation of CLOCK for read and write (CRAW). LRU: least recently used, MRU: most recently used.

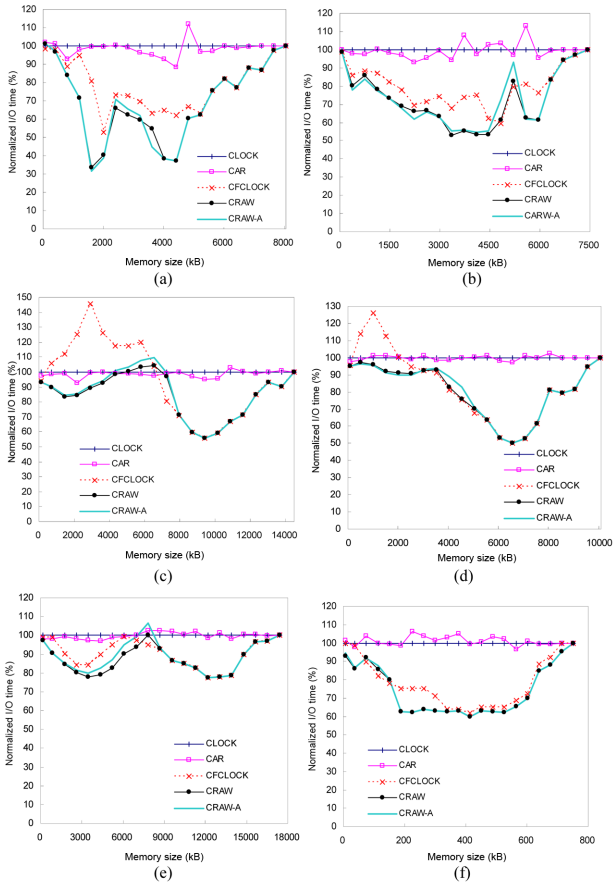

Fig. 17. Performance comparison of original Clock for read and write (CRAW) and its approximated algorithm CRAW-A. (a) xmms, (b) gqview, (c) gedit, (d) freecell, (e) kghostview, and (f) tar_gzip. CAR: Clock with adaptive replacement, CFCLOCK: .

within the handler, and the access permissions are changed. This is all done transparently to the application that actually accesses the memory. The kernel makes use of these bits when swapping pages in and out of memory; it is preferred to swap out pages that have not been recently used, and it is also ensured that pages that have been written to have their new contents copied to the backing store.

CRAW can be efficiently implemented in this environment as a software module by setting the read and the write bits through the simple modification of kernel similar to the way of abort mechanisms performed by the ARM architecture. This implies that CRAW can also be applied to the virtual memory of mobile embedded systems that does not support reference and modified bits for the paging unit hardware.

VII. CONCLUSIONS

Recently, NAND flash memory has been used as the swap space of virtual memory as well as the file system of embedded devices. Since temporal locality is dominant in page references of virtual memory, recency-based algorithms have been widely used. However, we showed that this is not the case for write references. We separately analyzed the characteristics of read and write references in virtual memory and found that the temporal locality of write references is weak and irregular. This implies that in the case of write operations, temporal locality alone is not sufficient to predict future references.

Based on this observation, we proposed and evaluated a new page replacement algorithm, called CRAW, that considers write frequency as well as temporal locality to predict the re-reference likelihood of write operations. CRAW separately analyzes the reference patterns of read and write operations depending on the characteristics of each operation, and more accurately predicts the re-reference likelihood of pages. To do this, CRAW partitions the memory space into a read area and a write area, and then dynamically adjusts their size according to the change of access patterns and the different I/O costs of read and write operations. Trace-driven simulations with various virtual memory access traces have shown that the proposed algorithm significantly improves the I/O performance of virtual memory systems. Specifically, it reduces I/O time by 20%–66% compared to widely known algorithms, such as CLOCK, CAR, and CFLRU.

In this paper, we focused on the temporal locality of read and write references. In future, we will analyze the inter-reference recency (IRR) property of read and write references [30]. We expect this to yield more interesting results. Various cost-aware algorithms will also be utilized. In this paper, we only considered the different I/O costs of read and write operations in terms of time among the various characteristics of NAND flash memory. We plan in the future to consider more specific characteristics of NAND flash memory.

AKNOWLEDGMENTS

This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korea government (No. 2011-0028825) and by the IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Devices). Hyokyung Bahn is the corresponding author of this paper.

REFERENCES

1. J. W. Hsieh, C. H. Wu, and G. M. Chiu, "MFTL: a design and implementation for MLC flash memory storage systems," *ACM Transactions on Storage*, vol. 8, no. 2, article no. 7, 2012.
2. Intel Corporation, *Understanding the Flash Translation Layer (FTL) Specification*, Denver: Intel Corporation, 1998.
3. W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: a file system for virtualized flash storage," *ACM Transactions on Storage*, vol. 6, no. 3, article no. 14, 2010.
4. A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of USENIX Technical Conference*, New Orleans, LA, 1995.
5. J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Transaction on Consumer Electronics*, vol. 48, no. 2, pp. 366-375, 2002.
6. O. Kwon, K. Koh, J. Lee, and H. Bahn, "FeGC: an efficient garbage collection scheme for flash memory based storage systems," *Journal of Systems and Software*, vol. 84, no. 9, pp. 1507-1523, 2011.
7. D. Woodhouse, "JFFS: the journaling flash file system," in *Proceedings of Ottawa Linux Symposium*, Ottawa, Canada, 2001.
8. YAFFS: Yet Another Flash File System, http://www.eleph1.co.uk/.
9. C. Park, J. U. Kang, S. Y. Park, and J. S. Kim, "Energy-aware demand paging on NAND flash-based embedded storages," in *Proceedings of International Symposium on Low Power Electronics and Design*, New Port, CA, 2004, pp. 338-343.
10. S. Y. Park, D. Jung, J. U. Kang, J. S. Kim, and J. Lee, "CFLRU: replacement algorithm for flash memory," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Seoul, Korea, 2006, pp. 234-241.
11. L. Shi, C. J. Xue, and X. Zhou, "Cooperating write buffer cache and virtual memory management for flash memory based systems," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Chicago, IL, 2011, pp. 147-156.
12. J. Park, H. Bahn, and K. Koh, "Buffer cache management

for combined MLC and SLC flash memories using both volatile and nonvolatile RAMs," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Beijing, China, 2009, pp. 228-235.
13. Intel Corporation, "Intel X-18M/X-25M SATA Solid State Drive (product manual)," http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf.
14. E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Englewood Cliffs: Prentice-Hall, 1973.
15. F. J. Corbato, *A Paging Experiment with the Multics System (MAC-M-384)*, Cambridge: MIT Press, 1969.
16. R. W. Carr and J. L. Hennessy, "WSCLOCK—a simple and effective algorithm for virtual memory management," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1981, pp. 87-95.
17. H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: integration of LRU and writes sequence reordering for flash memory," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 1215-1223, 2008.
18. H. Jo, J. U. Kang, S. Y. Park, J. S. Kim, and J. Lee, "FAB: flash-aware buffer management policy for portable media players," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 485-493, 2006.
19. S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, article no. 18, 2007.
20. H. Kim and S. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2008, pp. 239-252.
21. S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Transactions on Computers*, vol. 58, no. 6, pp. 744-758, 2009.
22. S. Bansal and D. S. Modha, "CAR: clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2004, pp. 187-200.
23. T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago de Chile, Chile, 1994, pp. 439-450.
24. Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, 2001, pp. 91-404.
25. N. Megiddo, and D. S. Modha, "ARC: a self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115-130.
26. E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, DC, 1993, pp. 297-306.
27. N. Nethercote and J. Seward, "Valgrind: a program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44-66, 2003.
28. Valgrind, http://valgrind.org/.
29. ARM, "Cortex-A series: programmer's guide," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013b.
30. S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: an effective improvement of the CLOCK replacement," in *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, 2005, pp. 323-336.

Hyejeong Lee

Hyejeong Lee received the B.S. degree in computer science and engineering from Ewha Womans University, Seoul, Republic of Korea, in 2006. She is currently an M.S. candidate in computer science and engineering, Ewha Womans University, Seoul, Republic of Korea. Her research interests include system security, operating systems, distributed systems, low power systems, intelligent storage systems, system optimization, ubiquitous computing, and embedded systems.

Hyokyung Bahn

Hyokyung Bahn received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a professor of computer science and engineering at Ewha Womans University, Seoul, Republic of Korea. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. He received the Best Paper Awards at the USENIX Conference on File and Storage Technologies in 2013. Prof. Bahn is a member of the IEEE Computer Society, the IEICE, and the KIISE.

Kang G. Shin

Kang G. Shin is the Kevin & Nancy O'Connor Professor of Computer Science in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive computing and networking as well as on embedded real-time and cyber-physical systems. He has supervised the completion of 74 PhDs, and authored/coauthored more than 800 technical articles (more than 300 of these are in archival journals), one textbook and more than 20 patents or invention disclosures, and received numerous best paper awards, including the Best Paper Awards from the 2011 ACM International Conference on Mobile Computing and Networking, the 2011 IEEE International Conference on Autonomic Computing, the 2010 and 2000 USENIX Annual Technical Conferences, as well as the 2003 IEEE Communications Society William R. Bennett Prize Paper Award and the 1987 Outstanding IEEE Transactions of Automatic Control Paper Award. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from The University of Michigan (the highest honor bestowed to Michigan Engineering faculty); a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering (the highest honor bestowed to Korean-origin engineers). He has chaired several major conferences, including 2009 ACM MobiCom, 2008 IEEE SECON, 2005 ACM/USENIX MobiSys, 2000 IEEE RTAS, and 1987 IEEE RTSS. He is the fellow of both IEEE and ACM, and served on editorial boards, including IEEE TPDS and ACM TECS. He has also served or is serving on numerous government committees, such as the US NSF Cyber-Physical Systems Executive Committee and the Korean Government R&D Strategy Advisory Committee. He has also co-founded a couple of startups.