MEMORY PAGE MANAGEMENT FOR PCM-BASED MOBILE EMBEDDED SYSTEMS

HYOKYUNGBAHN

EwhaWomans University, Seoul, Korea
E-mail: bahn@ewha.ac.kr

Abstract- Phase change memory is anticipated to be used as secondary storage in mobile embedded systems. These systems usually employ a compressed file system (CFS) to store system files which are fixed during the design phase, in combination with a normal file system to store data files. Since retrieving pages from a CFS requires additional decompression time, it is reasonable to grant them higher priorities when selecting an eviction victim. In this paper, we present a new page management policy for phase change memory based embedded systems that considers asymmetric operation cost of each page. The proposed algorithm considers the decompression cost of a page from CFS as well as the asymmetric I/O costs of reads and writes in PCM. To do this, the algorithm partitions the memory space into a read area, a write area, and a compressed area depending on different operation costs. The size of each area is then dynamically adjusted based on the change of access patterns and the contribution to reducing the I/O costs. Trace-driven simulations show that the proposed algorithm improves the I/O performance of mobile embedded systems significantly. Specifically, it reduces I/O time by 4.8-53.3% compared to widely acknowledged algorithms such as CLOCK, CAR, and CFLRU.

Index Terms- Compressed file system, Memory management policy, PCM, Storage.

I. INTRODUCTION

Phase change memory has emergedas a secondary storage mediumof modern embedded systems. These embedded systems usually employ compressed file systems such as CramFS [5] or SquashFS [4] to store system files which are fixed during the design phase, in combination with normal file systems that support both read and write operations for data files. By using compressed file systems, system designers can enjoy a larger effective storage space on the top of the limited storage capacity, and this makes a system more cost-effective. In addition, it is also helpful for system reliability, since compressed file systems are usually located in a read-only partition to prevent file corruption.

Despite its advantages of space efficiency and reliability, compressed file systems suffer from a performance degradation problem caused by the overhead of decompression at run time. To retrieve a page from a compressed file system, data should be fetched from the secondary storage and then decompressed, which is about seven times slower than a read from a normal file system [1]. This is an obstacle to the wide use of compressed file systems for an application in portable devices which requires fast response time. To relieve this problem, it is reasonable for page management policies to grant higher priorities to those pages from compressed file systems.

Another important problem that should be considered for the design of a page management policy is that the I/O cost of read and write operations in phase change memory is significantly asymmetric. Dirty pages need to be swapped out or flushed to phase change memory before eviction, and this incurs a write I/O which is about eight times slower than a read I/O. Note that a dirty page is a page that has been modified during its residence in the memory, while a clean page is a page whose contents have not been changed. Hence, an efficient management policy needs to take account of these asymmetric costs.

In this paper, we present a new page management policy for phase change memory based embedded systems that considers asymmetric operation cost of each page. Our policy considers the decompression cost of a page from a compressed file system as well as asymmetric I/O costs of read and write operations in PCM. To do this, the algorithm classifies page references as read, write, and compressed read, depending on their potential costs. Then, the algorithm partitions the memory space into three areas, namely read area, write area, and compressed area. The size of each area is then dynamically adjusted based on the change of access patterns and the contribution of each area to reducing I/O costs. In this process, ghost areas are adopted for each area to capture the workload changes. Ghost areas only maintain the metadata of a page evicted recently, and thus they are known to be lightweight [7, 8]. For each area, the recency of page references is separately captured using a CLOCK list. Trace-driven simulations show that the proposed algorithm, namely CRAW-C (CLOCK for read and write considering compressed file systems), improves the I/O performance of embedded systems significantly. Specifically, it reduces I/O time by 4.8-53.3% compared to widely acknowledged algorithms such as CLOCK, CAR, and CFLRU.

II. PAGE MANAGEMENT POLICY FOR PCM

A. System Architecture

In our system architecture,program files are stored in a read-only compressed file system such as CramFS,

while data files are stored in a normal file system such as JFFS2. Meanwhile, retrieving a page from the compressed file system requires additional time to decompress, and this procedure takes about seven times longer than retrieving a page from normal file systems. Therefore, replacing a page retrieved from the compressed file system and retrieving it again is relatively expensive.

B. Allocation and Adjustment of Each Area

CRAW-C allocates memory areas for read, write, and decompressed pages so as to minimize the total I/O costs. It does this by detecting the contribution of each area and adjusting their sizes dynamically. That is, read area manages recent read access to pages from normal file systems, write area manages recent write access to pages from normal file systems, and compressed area manages recent read access to pages from the compressed file system. All the pages in the write area are dirty pages which need about eight times higher cost in terms of time to evict than clean pages. Also, all the pages in the compressed area take about seven times longer to retrieve than pages from the normal file system. CRAW-C gives higher priorities to write pages and compressed 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 large.

CRAW-C employs ghost areas to evaluate and adjust the size of the read, write, and compressed areas as shown in Figure 1. Ghost areas contain the metadata of recently evicted pages without their actual data. By observing references to a page in the ghost areas, CRAW-C predicts the effect that extending each area would have on performance. If there are frequent hits on pages in the ghost read area, CRAW-C extends the read area to reduce the number of page faults. The write area and the compressed area are extended in the same way. In addition to the hits to pages in the ghost areas, the different cost of each access type is also considered in adjusting the size of each area. Specifically, read, write, and compressed areas are extended in proportion to their I/O costs 1:8:7.

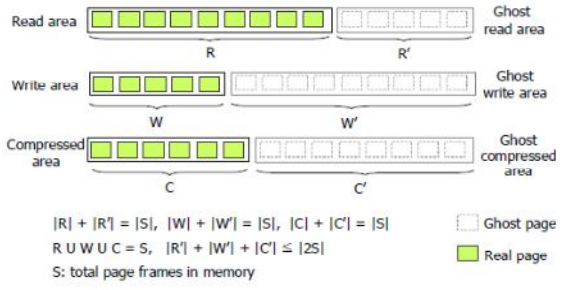


Figure 1.Adjusting the sizes of the read area, write area, and compressed area by using ghost areas.

different cost of each access type is also considered in adjusting the size of each area. Specifically, read, write, and compressed areas are extended in proportion to their I/O costs 1:8:7.

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-C shrinks the ghost read area by one. This is because the cache 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 Figure1, 2*S* ghost pages are sufficient for read, write, and compressed 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 4KB of data [7, 8].

Figure 1 shows the read area R, the write area W, the compressed area C, 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 manage the areas independently and accurately. Consequently, |R|+|W|+|C| can be larger than *S*, and the number of ghost pages can be equal to or less than 2*S*. Since CRAW-C allows pages to be linked to multiple areas simultaneously, a page can be evicted from physical memory only if it is not linked to any area.

C. Details of the Algorithm

Figure 2 depicts the basic structure of CRAW-C. Pages in memory are managed by read area R, write area W, and compressed area C. Metadata of evicted pages from these areas are kept in a corresponding ghost area for a while.

Page replacement for each area is managed independently by using an efficient design similar to the CLOCK algorithm. When CRAW-C searches for a victim to evict from the read area or the compressed area, it checks the read bit of the page to which the clock head points, as CLOCK also does. If the read bit is one, it is cleared; otherwise, the page is deleted from the area. The clock head scans through the pages clockwise until it finds a page with a zero read bit. When CRAW-C searches for a victim in the write area, the write bit is investigated instead of the read bit.In fact, the read and write bits have similar meanings to the reference and modified bits which are set by the paging unit hardware during every memory access.

The metadata for pages deleted from R, W, or C are inserted to the corresponding ghost area. When a replacement is needed in a ghost area, the least recently inserted page is deleted. As shown in Figure 2, a newly inserted page is linked to the MRU position in the ghost area, and the oldest page is evicted from the LRU position.

If the CPU references a page that is already in memory, CRAW-C 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 requested page is not in memory, CRAW-C first checks whether there is a free page frame. If not, CRAW-C 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. Then CRAW-C inserts the page in the area corresponding to the access and file system types, and adjusts the size of their areas if necessary.

If a page fault results from a read access, CRAW-C 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 the size of R is increased. The size of R' is then reduced to preserve the balance between R and R'. Page faults resulting from a write or read access which requires retrieval from the compressed file system are handled in the same way.

Now we will describe the reclamation procedure used to obtain a free page frame. Evicting a page from the physical memory is possible only if the page is not in R, W, or C. Therefore a single invocation of the *reclaim()* function may not be sufficient to obtain a free page frame. CRAW-C repeatedly invokes *reclaim()* until it obtains a free page frame. However, when CRAW-C evicts a page from the compressed area using the *reclaim()* function, the evicted page can always be discarded from the physical memory because a page in the compressed area cannot be linked to multiple areas, because it is not writable.

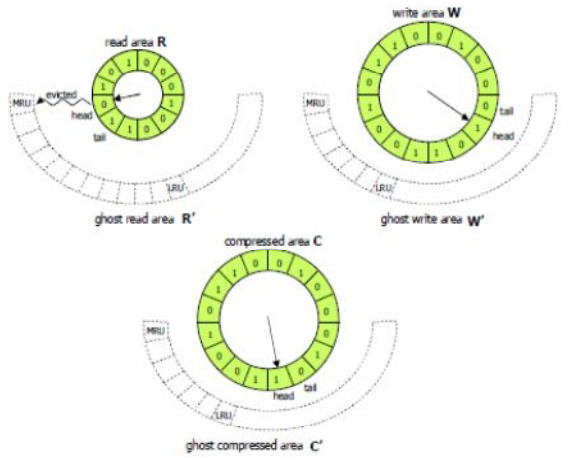


Figure 2.Basic structure of CRAW-C.

The*reclaim()* function first selects the area from which to evict a page. Since CRAW-C maintains and adjusts thedesired 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 satisfy this condition because CRAW-C allows a page to be shared by multiple areas. In this case, CRAW-C 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, W, and C are 3, 4, and 5, and current sizes of them are 4, 5, and 5, respectively. In this example, both R and W 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 or compressed areas, it first checks the read bit of the page to which the clock head points, as CLOCK does. If the read bit is one, *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 readarea, CRAW-C clears the write bit and links that page to the tail of write area W if the page is not in the write area already.If *reclaim()* selects write area 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, CRAW-C 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.

III. PERFORMANCE EVALUATION

Trace-driven simulation is performed to evaluate the management policy of a virtual memory system with accurate I/O timing of phase change memory including software overheads. The size of a page is set to 4KB which is common to most operating systems including Linux. Secondary storage consists of PCM. Read-only program files are placed in the compressed file system while other files including data files are placed in the normal file system.

The traces are acquired by a modified version of the Cachegrind tool from the Valgrind 3.2.3 toolset [2, 3]. We capture the virtual memory access traces from five different applications used on Linux Xwindows, namely the gedit word processor, the kghostview PDF file viewer, the freecell game, the xmms mp3 player, and the gqview image viewer. We filter out memory references that are accessed directly from the cache memory and also reflect the write-back property of the cache memory. The characteristics of these traces are described in Table 1.

To compare the performance of CRAW-C, we use CLOCK, CAR, and CFCLOCK [6, 7]. CFCLOCK is a modified version of CFLRU to work efficiently for virtual memory environments. On a hit to a page, CFCLOCK just sets reference bit or modified bit and does not perform list manipulations. Similar to CFLRU, CFCLOCK has a window size parameter that specifies

how many pages it should search for clean pages first, starting from the current head position. If it succeeds in finding a clean page with an access bit set to zero in the window, CFCLOCK replaces it. Otherwise, it scans through the pages in the window again and replaces a dirty page whose access bit is not set. If there is no such a page, then CFCLOCK searches for a page with an access bit set to zero, 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 one third of the total number of page frames according to the window size of swap system used in the original CFLRU simulations [6].

The performance of page management policies is measured by the total I/O time for the given workloads. Figure 3 shows the total I/O time for the fouralgorithms, with the memory size from 5% to 25% of maximum memory usage of the program. 100% memory size means the environment that a complete memory footprint can be loaded at the same time and thus no page replacement is needed. In this environment, the performances of all algorithms are identical.

Table 1. Memory characteristics for each workload trace.

Workload	Memory usage(MB)	Ratio of read : write
gedit	14.46	12.05 : 1
kghostview	17.39	13.93 : 1
freecell	10.08	7.16 : 1
xmms	8.05	1 : 5.13
gqview	7.43	1 : 1.30

Before comparing the performance of the algorithms, we briefly analyze the characteristics of the five traces in terms of the read/write intensity and their locality. Intensity of operations is provided in the third column of Table 1 and their locality can be seen from the trend of performance improvement as a function of the memory size in Figure 3.

- The xmmstrace is writeintensive.Note that the number of write references is five times more than the number of read references in the xmms trace. However,write I/O time is not relatively large compared to read I/O time as shown in Figure 3(d). This implies that most write references are concentrated to some hot pages which do not incur write I/Os because they are already resident in the memory even for the small memory size of 5%.
- Thegqview trace does not show intensity for read or write operation. However,the write locality of this trace is relatively strong as shown in Figure 3(e). Though the cost of a write I/O operation is larger than those of other operations, write I/O time is not relatively large in the figure.
- The gedit, freecell, and kghostviewtraces are read intensive and their read localityis also strong as the memory size increases from 5% to 25%. This is shown in Figures 3(a)-(c).

Now, let us compare the performance of the four algorithms. From the graphs in Figure 3, we can make the following observations.

- For all cases, the CRAW-C algorithm performs the best irrespective of the memory size, while the other algorithms give and take amongst each other at particular memory sizes and workloads. The reason is that CRAW-C adapts dynamically to changes of workload pattern and memory capacity. In comparison with CLOCK, CRAW-C reduced the total I/O time by 16.2% on average and by up to 53.3%. Also, CRAW-C outperforms CAR by 14.9% on average and by up to 53.1%, CFCLOCK by 20.1% on average and by up to 48.1%.

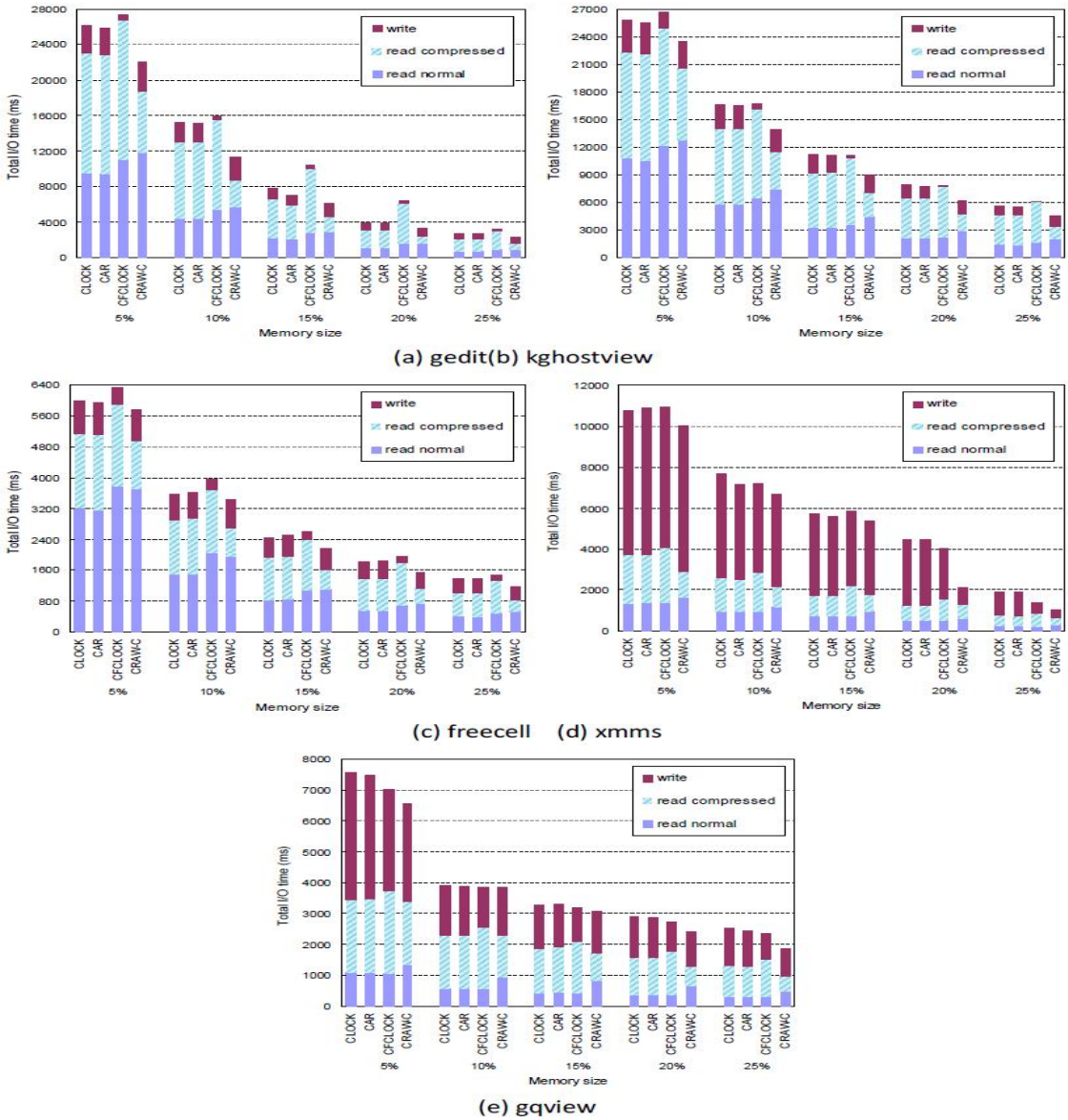


Figure 3. Total I/O time for phase change memory

•Since CFCLOCK prefers write references to read references irrespective of the relative intensity of read/write references, it does not perform well in read intensive workloads. Specifically, CFCLOCK performs even worse than CLOCK for gedit,freecell, and kghostview. Note that these three traces are read intensive and their locality is also strong. For all of these three traces, we can see the performance degradation of CFCLOCK in read and read compressed.

- Though xmms is write intensive, CFCLOCK does not perform well compared to CLOCK and CAR in some cases. This is because the xmms trace has some peculiar characteristics. Most write references are concentrated to a certain number of hot pages and they are already accommodated in the 5% memory size of all algorithms. Thus, the impact of write references is not large in the graphs. Specifically, the write I/O time of CFCLOCK is not sensitive to the memory size ranging from 5% to 20%.

- Thoughgqviewis not write intensive, it shows strong locality of write references during the change of memory size from 5% to 25%. In this trace, CFCLOCK reduces the number of expensive write I/O operations effectively by preserving dirty pages as much as possible, leading to improved performance compared to CLOCK and CAR.

•CAR performs better than CLOCK for most cases. Since CAR considers frequency as well as recency, it usually performs better than CLOCK that considers only the recency information. However, performance improvement of CAR is not evident in our experiments because temporal locality is dominant in virtual memory references.

CONCLUSION

Modern embedded systems usually employ compressed file systems to store program files in

Phase change memory. Since the cost of I/O operations in PCM is different according to the file system type (compressed or uncompressed) and I/O operation type (read or write), a cost-aware page management policy is needed. In this paper, we presented a new page management policy called CRAW-C for Phase change memory based embedded systems that considers asymmetric operation cost of each page. CRAW-C considers the decompression cost of a page from compressed file systems as well as asymmetric cost of read and writeoperations in PCM. CRAW-C partitions memory space into three areas, namely read area, write area, and compressed area depending on different operation costs. The size of each area is then dynamically adjusted based on the change of access patterns and the contribution of each area to reducing I/O costs. Through trace-driven simulations, we have shown that the performance improvement of CRAW-C against existing algorithms is in the range of 4.8-53.3%. Moreover, in contrast to LRU which needs list manipulations on every cache hit, CRAW-C does not need list manipulations unless page faults occur.

REFERENCES

[1] S. Hyun, H. Bahn and K. Koh, "LeCramFS: An Efficient Compressed File System for Flash-Based Portable Consumer Devices," IEEE Transactions on Consumer Electronics, vol. 53, no. 2, pp. 481-488, 2007.

[2] http://valgrind.org/

[3] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," Electronic Notes in Theoretical Computer Science, 2003.

[4] SquashFS, http://squashfs.sourceforge.net/

[5] CramFS, http://lxr.linux.no/source/fs/cramfs/README

[6] S. Park, D. Jung, J. Kang, J. Kim and J. Lee, "CFLRU: management policy for PCM," Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, 2006.

[7] S. Bansal and D. S. Modha, "CAR: clock with adaptive replacement," Proceedings of the 3rd USENIX Conference on File and Storage Technologies, 2004.

[8] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management management policy," Proceedings of the VLDB conference, 1994.

★★★