International Journal of Advanced Smart Convergence Vol.12 No.1 70-75 (2023)
http://dx.doi.org/10.7236/IJASC.2023.12.1.70

IJASC 23-1-9

Considering Read and Write Characteristics of Page Access Separately for Efficient Memory Management

Hyokyung Bahn

Professor, Department of Computer Engineering, Ewha University, Professor, Korea
bahn@ewha.ac.kr

Abstract

With the recent proliferation of memory-intensive workloads such as deep learning, analyzing memory access characteristics for efficient memory management is becoming increasingly important. Since read and write operations in memory access have different characteristics, an efficient memory management policy should take into account the characteristics of these two operations separately. Although some previous studies have considered the different characteristics of reads and writes, they require a modified hardware architecture supporting read bits and write bits. Unlike previous approaches, we propose a software-based management policy under the existing memory architecture for considering read/write characteristics. The proposed policy logically partitions memory space into the read/write area and the write area by making use of reference bits and dirty bits provided in modern paging systems. Simulation experiments with memory access traces show that our approach performs better than the CLOCK algorithm by 23% on average, and the effect is similar to the previous policy with hardware support.

Keywords: *Memory management, paging system, read, write, memory reference, CLOCK.*

1. Introduction

Due to the recent proliferation of memory-intensive workloads such as big data processing and deep learning, analyzing page access characteristics for efficient memory management is becoming increasingly important [1]. Page access is known to have strong temporal locality, implying that a recently accessed page is highly likely to be accessed again [2]. To utilize this property, the CLOCK algorithm and its variants are widely adopted as memory management policies [3, 4]. However, such policies simply keep recently accessed pages in memory without distinguishing read and write operations. Some recent studies try to utilize read and write references separately for further improving the efficiency of memory management [3, 5]. This is also related to the emergence of storage media such as flash memory and NVRAM, which have asymmetric read/write access latency [5, 6]. Although NVRAM can also be adopted in memory layers due to its byte addressable features, it is more attractive as high performance storage medium because it is relatively slower than DRAM [7, 8].

Manuscript Received: January. 5, 2023 / Revised: January. 8, 2023 / Accepted: January. 11, 2023
Corresponding Author: bahn@ewha.ac.kr
Tel: +82-2-3277-2368, Fax: +82-2-3277-2306
Professor, Department of Computer Engineering, Ewha University, Korea

As flash memory and NVRAM have asymmetric read/write latency, an efficient memory management policy needs to take into account the relative performance of each operation. From this point of view, pages residing in memory can be categorized as clean pages and dirty pages. A clean page is a page that has only been read after loaded into memory. As the original data is also maintained in storage, it can be simply discarded when a clean page is evicted from memory. On the other hand, a dirty page is a page that has been written at least once after being loaded into memory; when such a page is evicted from memory, it should first be reflected to storage. In other words, since a dirty page accompanies a write operation to storage, it is necessary to raise the in-memory priority of dirty pages when slow-write media such as flash memory and NVRAM are adopted [3].

Meanwhile, it has been observed that reads and writes of memory pages have different access characteristics [2, 3]. In read access, temporal locality is strong, so there is a high possibility of re-accessing recently read pages, whereas temporal locality of write access is relatively weak and irregular. Although some memory management policies consider these different characteristics of reads and writes, their assumption is that read and write accesses can be separately tracked by hardware assistance [3]. However, it is not easy to identify read and write operations from each page access in memory systems. This is because only a bit setting is performed by hardware in each memory access, from which we cannot recognize whether reading or writing occurs. Specifically, memory management hardware sets the reference bit of a page to 1 whenever the page is accessed regardless of reading or writing. When the access is write, a dirty bit of the page is also set to 1. Therefore, in order to distinguish an operation type for each memory access, modifications in hardware architecture are required.

In this article, we discuss how to take into account the different characteristics of read and write operations in designing a memory management policy without modifying hardware architectures. Previous studies have argued that a hardware architecture with read bits and write bits is necessary to utilize the characteristics of read/write operations. Lee et al. propose separate settings of memory access bits for different operation types to exploit the access characteristics of reads and writes in memory management [3]. Park et al. argue that distinguishing instruction and data access as well as reads and writes is necessary for further improving memory system performance as the cost of accessing instruction and data storage is different [9]. However, all of these studies assume architectural support for bit settings in their memory management policies. In contrast, this article suggests an alternative method that exploits reference bits and dirty bits provided by the current hardware architecture. For example, the Intel x86 architecture sets the dirty bit of a page table entry when there is the first write access to a memory page; our approach does not need the modification of such architectures because we use the supported dirty bits as-is, differentiated from previous studies. Through simulation experiments by replaying memory access traces of various workloads, we show that the proposed method performs better than the CLOCK algorithm by an average of 23%. In addition, we show that the effect of the proposed method is almost identical to the result that requires the support of hardware architecture.

2. The CLOCK and CRAW Algorithms

It is known that page access in virtual memory systems has strong temporal locality [3]. Temporal locality refers to the property that a more recently accessed page is more likely to be accessed again in the near future. For access characteristics with temporal locality, the LRU (least recently used) algorithm is known to be optimal [10]. The LRU algorithm discards the oldest accessed page from memory when free memory space is necessary. The LRU algorithm uses a linked list to sort pages in memory according to their access order and whenever a memory access occurs, the accessed page moves to the highest priority position in the list. Since

LRU is simple to implement, it is widely used in various computing environments such as file caching and web caching [11, 12]. However, in a memory system, it is almost impossible to update the access time information of a page whenever it is accessed. This is because the operating system kernel cannot perform list manipulations for all memory accesses. It is also difficult to maintain the time information of each memory access with hardware support. Therefore, in the current memory system architecture, the CLOCK algorithm is used as an approximation of LRU [4]. When a memory page is accessed, the reference bit of the page is set to 1 instead of maintaining the exact access time. Based on these reference bits, CLOCK sequentially scans all pages to search pages that have not been recently accessed when free memory space is needed. During this scan, if a memory page with the reference bit set to 1 is found, it is reset to 0; otherwise, if a page whose reference bit is 0 is found, it is discarded.

The CRAW (CLOCK for read and write) algorithm has been proposed as a memory management policy that exploits read and write references separately [3]. CRAW manages the memory area by dividing it into a read area and a write area to consider the asymmetric cost of read/write operations when using flash memory as a virtual memory swap device. In order to minimize the I/O cost of the flash memory, the contributions of the read and write areas are investigated and the sizes of the two areas are dynamically adjusted. CRAW selects victim pages in each area like CLOCK, but read and write bits are used instead of the reference bit to separately utilize the characteristics of read/write references. To compensate for the weak temporal locality of write references, the write frequency is used together in the write area. Since a write operation to flash memory costs 2 to 8 times higher than a read operation, CRAW assigns higher priority to pages in the write area than the read area. CRAW maintains two history lists, a read history list and a write history, to monitor the impact of the read/write areas and adjusts them accordingly. The history lists temporarily preserve the metadata of a page evicted from memory to indicate that the page has recently been released. Through the page access information in the history lists, the performance effect when the size of the area is enlarged can be expected. For example, if pages belonging to the read history list are frequently referenced, we can improve performance by enlarging the read area. CRAW adjusts the size of the areas by considering not only page hits in the history lists but also the cost of an operation that occurs during storage I/Os.

Figure 1(a) shows the circular linked list managed by CLOCK, and Figure 1(b) shows the read and write areas, and their history lists managed by CRAW. When read and write references occur to the same page, CRAW maintains that page in both read and write areas simultaneously. Since one page can exist in multiple areas, a page can be discarded from the physical memory only when it is not connected to any area.

3. Considering Read/Write Characteristics in Memory Management

Previous studies exploiting the read/write characteristics of memory access assume that the two operations can be captured separately by the memory management system. For example, CRAW assumes that read/write bits can be set when a read/write reference occurs. Although these two bits are similar to the reference bit and the dirty bit in the current paging system, they are not supported by the existing memory management hardware. Thus, in order to realize memory management with the separation of reads and writes, hardware modifications is necessary.

This article aims to exploit the different characteristics of read and write references at the operating system layer without the modification of hardware architecture. That is, we only make use of the reference bit and the dirty bit provided by the existing architecture. As it is not possible to separate read access from the two existing

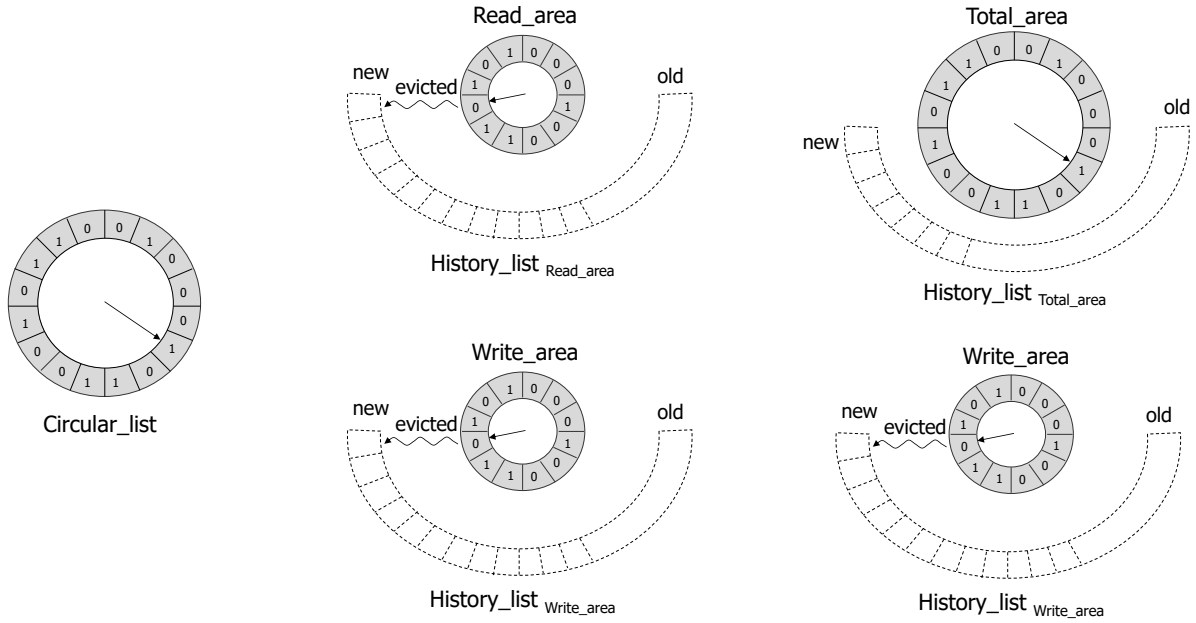

Figure 1. Comparison of data structures used in CLOCK, CRAW, and the proposed policy. CLOCK uses a single circular list; CRAW uses two circular lists for read and write areas along with their history lists; the proposed policy uses two circular lists for total and write areas along with their history lists.

bits (i.e., reference bit and dirty bit), the only way we can pursue is to extract read and write accesses together according to the semantics provided by the reference bit and the dirty bit. That is, since both reading and writing cause the setting of the reference bit, the overall access characteristics are extracted through the reference bit of the page, and the characteristics of write access can be captured by making use of the dirty bit.

Just as CRAW uses the read bit and the write bit to manage the read area and the write area, respectively, the proposed policy manages the total area (i.e., read+write area) and the write area by utilizing the reference bit and the dirty bit, respectively. In order to accurately reflect the meaning of the bits, when a write access occurs to a certain page, we add it to both the total area and the write area. This allows for determining the size of the two areas accurately based on the exact contribution of each area with respect to the two operations. This is reasonable as the temporal locality considering both read and write accesses shows a similar distribution to the temporal locality of read access only [3]. In contrast, the temporal locality of write access is very different, and this is well taken into account in our policy by managing a separate write area. Figure 1(c) briefly shows the data structure of the proposed policy in comparison with CLOCK and CRAW.

4. Simulation Experiments

To evaluate the performance of the proposed policy, we conduct simulation experiments by replaying the memory access traces. Our trace consists of four workloads on Linux: kghostview, freecell, xmms, and geeqie [13]. Figure 2 shows the execution time of the three policies as the memory size is varied. Note that the

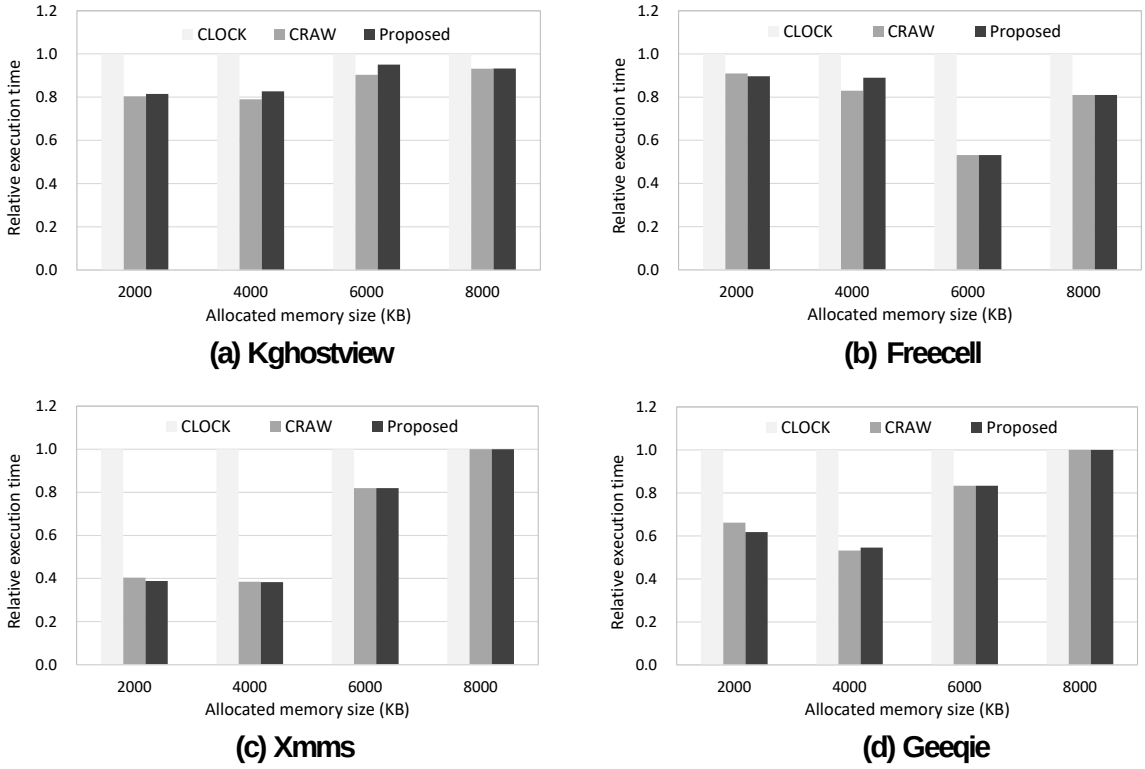

Figure 2. Comparison of the execution time in CLOCK, CRAW, and the proposed policy as the allocated memory size is varied for the four workloads we experiment.

execution time is expressed as a relative value of the CLOCK's result. As shown in the figure, CRAW and the proposed policy show superior performance compared to CLOCK, and the improvement is evident when the memory size is small with relatively write-intensive workloads such as xmms and geeqie. When the memory size is large enough to accommodate the entire workload, pages need not be evicted from memory regardless of any policies used and the performance converges to the same result. Conversely, when the memory size is small, the performance depends on how judiciously the algorithm works except for some extremely small memory cases where it is hard to make a difference in performance. The performance improvement of the proposed policy against CLOCK is 23% on average. Though the proposed policy does not need hardware supports, its result is similar to CRAW, which requires the assistance of read/write bits.

5. Conclusion

Since read and write operations in memory accesses have different characteristics, an efficient memory management policy should utilize the two operations separately. While previous studies required hardware modifications for this purpose, we proposed a software-based management policy under the existing memory architecture for taking into account the read/write characteristics. Specifically, the proposed policy logically partitions memory space into the total area and the write area by making use of reference bits and dirty bits provided in modern paging systems. Simulation experiments with memory access traces showed that our approach performed better than the CLOCK algorithm by 23% on average, and the effect was similar to the previous policy that needs hardware support.

Acknowledgement

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C1009275) and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-02068, Artificial Intelligence Innovation Hub).

References

[1] S. Dargan, M. Kumar, M.R. Ayyagari and G. Kumar, "A survey of deep learning and its applications: a new paradigm to machine learning," Archives of Computational Methods in Engineering, vol. 27, no. 1, pp. 1071–1092, 2020. DOI: https://doi.org/10.1109/TC.2013.98

[2] S. Lee, H. Bahn, and S. H. Noh, ''CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures,'' IEEE Trans. Computers, vol. 63, no. 9, pp. 2187-2200, 2014. DOI: https://doi.org/10.1109/TC.2013.98

[3] H. Lee and H. Bahn, "Characterizing virtual memory write references for efficient page replacement in NAND flash memory," Proc. IEEE MASCOTS Conf., pp.1-10, 2009. DOI: https://doi.org/10.1109/MASCOT.2009.5366768

[4] F. J. Corbato, "A paging experiment with the Multics system," In Honor of Philip M. Morse, 1st ed., Cambridge, MA, USA, The MIT Press, pp. 217-228, 1969.

[5] S. Lee and H. Bahn, "Characterization of Android memory references and implication to hybrid memory management," IEEE Access, vol. 9, pp. 60997-61009, 2021. DOI: https://doi.org/10.1109/ACCESS.2021.3074179

[6] J. Yue and Y. Zhu, "Accelerating write by exploiting PCM asymmetries," Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 282-293, 2013. DOI: https://doi.org/10.1109/HPCA.2013.6522326

[7] S. Yoo, Y. Jo, and H. Bahn, "Integrated scheduling of real-time and interactive tasks for configurable industrial systems," IEEE Trans. Industrial Informatics, vol. 18, no. 1, pp. 631-641, 2022. DOI: https://doi.org/10.1109/TII.2021.3067714

[8] S. Yoon, H. Park, K. Cho, and H. Bahn, "Supporting swap in real-time task scheduling for unified power-saving in CPU and memory," IEEE Access, vol. 10, pp. 3559-3570, 2022. DOI: https://doi.org/10.1109/ACCESS.2021.3140166

[9] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn, "A cost-aware page replacement algorithm for NAND flash based mobile embedded systems," Proc. ACM EMSOFT Conf., pp. 315-324, 2009. DOI: https://doi.org/10.1145/1629335.1629377

[10] E. Coffman and P. Denning, Operating Systems Theory, Prentice-Hall, pp. 241-283, 1973.

[11] H. Bahn, S. Noh, S. Min, and K. Koh, "Efficient replacement of nonuniform objects in web caches," Computer, vol.35, no.6, pp.65-73, 2002. DOI: https://doi.org/10.1109/MC.2002.1009170

[12] O. Kwon, H. Bahn, and K. Koh, "Popularity and prefix aware interval caching for multimedia streaming servers," Proc. IEEE Conf. on Computer and Information Technology, pp. 555-560, 2008. DOI: https://doi.org/10.1109/CIT.2008.4594735

[13] Memory access traces, https://github.com/oslab-ewha/memtrace