Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 109–117.

DesynchLRU: An Efficient Page Replacement Algorithm with Desynchronized Cache and RAM*

Md. Raqibul Hasan, M. Sohel Rahman, Chowdhury Sayeed Hyder

Department of Computer Science and Engineering Bangladesh University of Engineering and Technology Dhaka-1000, Bangladesh
raqib_cse@yahoo.com, msrahman@cse.buet.ac.bd, sayeed@csebuet.org

Abstract

In this paper, we present DesynchLRU, a new page replacement algorithm. DesynchLRU is a modified version of the celebrated LRU (Least Recently Used) page replacement algorithm. The basic distinction between LRU and DesynchLRU is that, in the former, the pages in cache and RAM always remain synchronized, whereas, in the latter, they can be desynchronized. Here, we show that the page fault rate in DesynchLRU is always smaller than that of LRU. We further analyze the average cost for a page request and establish conditions for which DesynchLRU would always perform better than LRU.

Keywords: Virtual memory, page replacement, main memory, cache, page fault.

1. Introduction

With the advent of newer and faster technologies, more and more faster memory chips are being built and various frameworks with different levels of memory hierarchy are being presented and used in computing systems. All these sophisticated systems however are extensions of the basic 2-level memory hierarchy system, where the 2 levels are formed by main memory (i.e., RAM) and the virtual memory [4, 5] (i.e., disk). The motivation for such levels of memory hierarchy is well-known and is briefly described as follows. The combined size of the program, data and

*The work was carried out under the framework of EU funded eLINK project (eLINK – east-west Link for Innovation, Networking and Knowledge exchange Programme ECW- 149674-UK-L12_2008-4949).

stack may exceed the physical memory available for it. Therefore, the operating system keeps only those parts of the program currently in use in main memory. Clearly, the idea is to optimize the speed and cost of the program execution by using more costly but faster main memory along with the cheaper but slower disk as virtual memory. The basic model works as follows. The programs are divided in blocks referred to as pages henceforth. The main memory contains the pages of the program currently in use whereas the disk contains all the pages of the program. Therefore, main memory contains a subset of pages in the disk. When a page is referenced, that is not present in the memory, a page fault occurs. When a page fault occurs, the operating system fetches the corresponding page from the disk and has to replace (evict) a page from the main memory to make space for the fetched page. Now, there are two issues to be considered here. The first issue relates to which page is to be evicted from the main memory: this is decided by the page replacement algorithms. The second issue is that the page to be replaced must be copied back to the corresponding page in the disk if the former is changed to keep that up to date. This procedure is referred to as writeback. To avoid redundant time-consuming disk write, writeback is performed only when the page in the memory is changed (the page is said to be dirty) and selected for eviction.

In the memory hierarchy, position of cache is between processor and main memory. Cache memories [12, 15, 16] are used in modern, medium and high-speed CPUs to hold temporarily those pages of main memory which are (believed to be) currently in use. Cache is faster but costlier than RAM. The same relation that holds between RAM and disk, holds between Cache and RAM. If the requested page is not present in cache, this event is called a cache miss. When a cache miss occurs, the operating system has to choose a page from cache for eviction (using a page replacement algorithm) to make room for the page that has to be brought in. If the selected page is modified while in cache (i.e. dirty), it must be rewritten to the RAM (i.e. a writeback must occur). If, however, the page has not been modified, no writeback is needed. Note that, if both cache miss and RAM miss occur, the page from the disk is first brought to RAM and then the same is brought to cache. In this paper, we focus on page replacement algorithms. The issue of page fault and the corresponding page replacement algorithms have been the focus of tremendous attention in operating systems research [1-3, 6-11, 13, 14, 18]. While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Among the existing page replacement algorithms in the literature, Least recently used (LRU) is the most popular and simple. Despite many replacement algorithms proposed throughout the years, LRU approximations are predominant in case of actual virtual memory management systems. LRU, however, exhibits well-known performance problems for regular access patterns over more pages than the main memory can hold (e.g., large loops).

Again, most of the algorithms do not consider the access time of cache and RAM (i.e. variation of speed). For example, reading a page from cache needs lesser time than reading from RAM. Similarly, disk access time is much greater than memory access time. In this paper, we present a variation of traditional LRU algorithm considering these facts. Unlike traditional system, our algorithm desynchronizes the content of cache and memory. We believe that our algorithm will reduce page fault rate and also reduce the page replacement cost to some extent. The rest of the section is organized as follows. In Section 2 we briefly review the related page replacement algorithms. We present our main result in Section 3. Finally, we briefly conclude in Section 4.

2. Page Replacement Algorithms

The best possible page replacement algorithm is easy to describe but impossible to implement [17]. The idea is as follows. At the moment that a page fault occurs, some sets of pages are in memory. Now, one of these pages will be referenced on the very next instruction (the page containing that instruction). So, we can label each page with the number of instructions that will be executed before that page is first referenced. Assuming that we can do the above, the optimal page replacement algorithm simply says that the page with the highest label should be removed. The only problem however is that this algorithm is not realizable, because, at the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement algorithm on the second run by using the page reference information collected during the first run.

2.1. LRU Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called Least Recently Used (LRU) paging page replacement algorithm.

3. Our approach

In this section, we present a new approach for page replacement. The basic distinction of our approach with the existing page replacement algorithms is that in the latter, the pages in cache and RAM always remain synchronized, whereas, in the former, they can be desynchronized. In the rest of this section, we describe our strategy and present DesynchLRU, the new page replacement algorithm.

3.1. Assumption

An important issue with respect to the page replacement algorithms is to decide what will be the cache block and main memory page size and whether they be of equal size? If they are of different size (say two block is equal to one page), then for a block in cache we have to keep track of its main memory page number and block number in that page. Moreover complexity will arise for cache writeback. So, in most of the systems, cache block and memory page are of equal size to keep memory management task simple. Here, as well, we shall consider system with cache block and memory page of equal size.

3.2. DesynchLRU: The New Algorithm

In LRU, the set of pages in cache is a subset of pages in the main memory. So, if a page is in the cache, then it will also be in the main memory, i.e., the main memory and the cache is always synchronized with each other. In DesynchLRU, the set of pages in cache may not be a subset of pages in main memory i.e. if a page is in cache its copy may not be in the main memory. Therefore, the cache and the main memory will be desynchronized. The main idea of DesynchLRU is as follows. When selecting a page from main memory (RAM) for eviction, DesynchLRU tries to select a page whose copy is present in cache. If such a page is not available then it uses LRU algorithm. If cache page fault occurs, then a page from cache must be replaced. In case of LRU, we have to write back the page in the main memory only if the selected page is dirty; the writeback is not needed if the page is not dirty. But in DesynchLRU, the image of the selected page may not be present in the main memory. So, it may need to select a page from main memory for eviction to writeback the page selected from cache irrespective of whether it is dirty or not.

Cache	RAM	Action
Hit	Hit	Nil
Hit	Miss	Nil
Miss	Hit	-Select a page P_c from cache for eviction according to LRU policy. -Writeback P_c in RAM evicting a page from RAM. - Bring the page in cache which caused cache miss.
Miss	Miss	-Select a page from RAM for eviction. -Bring the page in RAM from disc. -Select a page P_c from cache for eviction. -Writeback P_c in RAM evicting a page from RAM. - Bring the page in cache which caused cache miss.

Table 1: Different scenario of updating cache and RAM

Lemma 3.1. *In normal state there will always exist at least one page which is both in cache and RAM.*

Proof. Consider that the last page fault PF_last in the cache was for page p. Recall that, to bring a page in cache, we first bring it in the RAM from disk and then bring it in the cache from the RAM. So, after we handle PF_last page p exist both in cache and RAM. Since, PF_last was the last page fault in the cache and since page p from RAM can never be evicted without a further page fault in the cache, the result follows. □

3.3. Page Fault in RAM for DesynchLRU

In this subsection, we analyze the probability of page faults in RAM in strategy DesynchLRU. Assume that PF_DesynchLRU and PF_LRU denote the probability of page fault in RAM, respectively, for DesynchLRU and LRU. We prove the following lemma.

Lemma 3.2. *PF_DesynchLRU < PF_LRU*

Proof. Assume that there are *k, m* and *n* pages, respectively, in the cache, RAM and the disk. In case of LRU, the probability that a virtual page is in cache or RAM is $\frac{m}{n}$. This follows from the fact that in LRU, Cache is synchronized with RAM. Therefore, the RAM page fault can only occur when a page p is referenced such that p doesn't exist in RAM. So, PF_LRU $= 1 - \frac{m}{n}$. In case of DesynchLRU, the probability that a virtual page is in cache or RAM is $\frac{m+k}{n}$. Now, recall that in this case, Cache is not synchronized with RAM anymore. Therefore, we have PF_DesynchLRU $= 1 - \frac{m+k}{n}$. Clearly, $1 - \frac{m+k}{n} < 1 - \frac{m}{n}$ and hence the result follows. □

3.4. An Illustrative Example

Lemma 2 proves that the page fault rate of DesynchLRU is lesserthan that of LRU. We now present an example illustrating thatfact. Consider a cache with 3 pages, a main memory with 6 pagesand the reference string 1 2 3 4 5 6 7 8 1 2. The situations with LRU and DesynchLRU are illustrated in Figure 1 and 2 respectively.

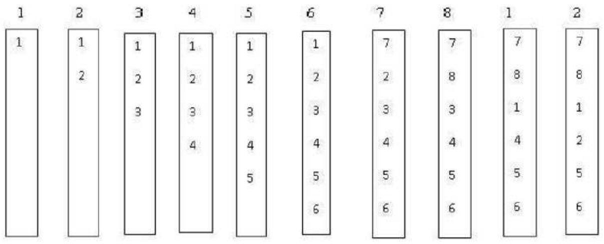

Figure 1: In case of LRU

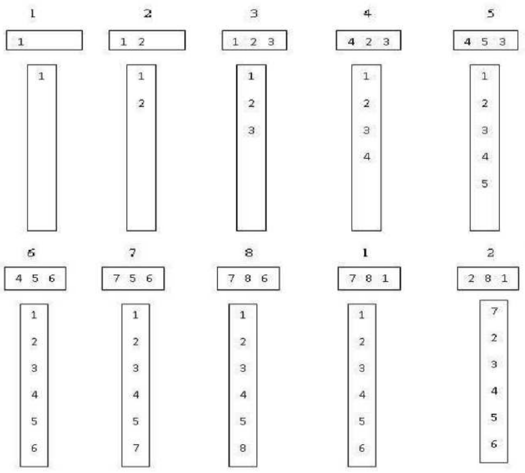

Figure 2: In case of DesunchLRU

Remark 3.3. Here we can observe that, in case of DesynchLRU, there is no main memory page fault for pages 1 and 2 when they are referenced for the second time. But in case of LRU page fault occurs.

3.5. A Probabilistic Analysis

Previously we have shown that probability of page fault in our proposed algorithm is less than that of LRU (Lemma 3.2). But this criterion alone does not always justify the superiority of an algorithm. In case of LRU, when we evict a page from cache we only writeback the page in RAM if the page is modified (dirty) while in cache. But in case of the proposed algorithm when we evict a page from cache we may need to writeback the page in RAM irrespective of whether it is dirty or not. This may create an extra overhead. In the rest of this subsection, we take an effort to deduce the exact condition for which DesynchLRU will perform better than LRU.

In case of LRU cost of page request in different scenario of RAM and cache are as follows:
Cache - hit,RAM -hit: no cost
Cache -hit, RAM -miss: this scenario will not occur in LRU
Cache -miss, RAM -hit:

- Cost to transfer the requested page from RAM to cache.
- If the evicted page from cache is dirty then the writeback cost of cache to RAM for that page.

Cache -miss, RAM -miss:

- Cost to transfer the requested page from RAM to cache.
- If the evicted page from RAM is dirty then the writeback cost of RAM to disk for that page.

– then the situation is as like Cache -miss, RAM -hit.

In case of DsynchLRY cost of page request in different scenario of RAM and cache are as follows:

Cache - hit,RAM -hit: no cost

Cache -hit, RAM -miss: no cost

Cache -miss, RAM -hit:

– Cost to transfer the requested page from RAM to cache.

– writeback cost of cache to RAM for the evicted page from cache.

Cache -miss, RAM -miss:

– select a page from cache for eviction to make room for the requested page thus writeback cost of cache to RAM for the evicted page.

– cost to transfer the requested page from disk to RAM.

– If the evicted page from RAM is dirty and the page is not in cache then the writeback cost of RAM to disk for that page.

– cost to transfer the page from RAM to cache.

Let, the number of pages in cache, RAM and disk are respectively *k, m* and *n*. Assume that the probability that a page is dirty, P(dirty) $= \delta$. We further assume that the costs of transferring a page from RAM to cache and RAM to disk are, respectively, α and β. Let C_LRU and C_DesynchLRU denote the average cost for an arbitrary page request, respectively, for LRU and DesynchLRU. Since, it is not possible to assume exact page reference pattern before executing the concerned program, we perform an average case analysis. Here, we assume uniform probability for each page to be referenced. In case of LRU

P(*cache hit, RAM hit*) $= \frac{k}{n}$. [As if a page is in cache it is also in RAM]

P(*cache miss, RAM miss*) $= 1 - \frac{m}{n} = \frac{n-m}{m}$. [As if a page is not in RAM it can not be present in cache]

P(*cache miss, RAM hit*) = 1 - P(*cache hit, RAM hit*) - P(*cache miss, RAM miss*) $= \frac{m-k}{n}$

$$\begin{aligned} \text{C\_LRU} &= 0 \cdot \tfrac{k}{n} + \tfrac{m-k}{n}[\alpha + \delta\alpha] + \tfrac{n-m}{n}[\beta + \delta\beta + \alpha + \delta\alpha] \\ &= \alpha\tfrac{n-k}{n} + \alpha\delta\tfrac{n-k}{n} + \beta\tfrac{n-m}{n} + \beta\delta\tfrac{n-m}{n}. \end{aligned}$$

In case of the proposed algorithm (here we can consider pages in cache and RAM are independent).

P(*cache hit, RAM hit*) $= \frac{k}{n} \cdot \frac{m}{n}$

P(*cache hit, RAM miss*) $= \frac{k}{n} \cdot \frac{n-m}{n}$

P(*cache miss, RAM miss*) $= \frac{n-k}{n} \cdot \frac{n-m}{n}$

P(*cache miss, RAM hit*) = 1 - P(*cache hit, RAM hit*) - P(*cache hit, RAM miss*) - P(*cache miss, RAM miss*) $= \frac{m(n-k)}{n^2}$

Average cost for an arbitrary page request is

C_DesynchLRU

$$\begin{aligned}
&= 0 \cdot \tfrac{k}{n} \cdot \tfrac{m}{n} = 0 \cdot \tfrac{k}{n} \cdot \tfrac{n-m}{n} + \tfrac{m(n-k)}{n^2}[\alpha + \alpha] + \tfrac{(n-k)(n-m)}{n^2}\left[\beta + \tfrac{n-k}{n}\delta\beta + \alpha + \alpha\right] \\
&= 2\alpha\left[\tfrac{m(n-k)}{n^2} + \tfrac{(n-k)(n-m)}{n^2}\right] + \beta\tfrac{(n-k)(n-m)}{n^2} + \beta\delta\tfrac{(n-k)^2(n-m)}{n^3} \\
&= 2\alpha\tfrac{n-k}{n} + \beta\tfrac{(n-k)(n-m)}{n^2} + \beta\delta\tfrac{(n-k)^2(n-m)}{n^3}.
\end{aligned}$$

Remark 3.4. $\frac{n-k}{n}\delta\beta$ is the writeback cost of RAM to disk for the evicted page from RAM. This cost is incurred when the evicted page from RAM is dirty and the page is not in cache. Now, for DesynchLRU to outperform LRU we must have C_LRU > C_DesynchLRU. So, we have:

$$\begin{aligned}
&\alpha\tfrac{n-k}{n} + \alpha\delta\tfrac{n-k}{n} + \beta\tfrac{n-m}{n} + \beta\delta\tfrac{n-m}{n} > 2\alpha\tfrac{n-k}{n} + \beta\tfrac{(n-k)(n-m)}{n^2} + \beta\delta\tfrac{(n-k)^2(n-m)}{n^3} \\
&\Rightarrow \delta\left[\alpha\tfrac{n-k}{n} + \beta\tfrac{n-m}{n} - \beta\tfrac{(n-k)^2(n-m)}{n^3}\right] > \alpha\tfrac{n-k}{n} + \beta\tfrac{(n-k)(n-m)}{n^2} - \beta\tfrac{n-m}{n} \\
&\Rightarrow \delta > \frac{\alpha\frac{n-k}{n} + \beta\frac{(n-k)(n-m)}{n^2} - \beta\frac{n-m}{n}}{\alpha\frac{n-k}{n} + \beta\frac{n-m}{n} - \beta\frac{(n-k)^2(n-m)}{n^3}}
\end{aligned}$$

4. Conclusion

In this paper, we have presented DesynchLRU a new page replacement algorithm, which is essentially a modified version of the celebrated LRU algorithm. The basic distinction of our approach with the existing page replacement algorithms is that in the latter, the pages in cache and RAM always remain synchronized, whereas, in the former, they can be desynchronized. We have shown (in Lemma 3.2) that the page fault rate in DesynchLRU is smaller than that of LRU and hence we are not giving any simulation result. We have also analyzed the average cost for a page request and establish conditions for which DesynchLRU would always perform better than LRU.

References

[1] Kunal Agrawal, Michael A. Bender, and Jeremy T. Fineman. The worst pagereplacement policy. In FUN, pages 135-145, 2007.

[2] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. J. ACM, 18(1):80-93, 1971.

[3] Jaafar Alghazo, Adil Akaaboune, and Nazeih Botros. Sf-lru cache replacement algorithm. In MTDT, pages 19-24. IEEE Computer Society, 2004.

[4] A. Bensoussan, C. T. Clingen, and Robert C. Daley. The multics virtual memory: Concepts and design. Commun. ACM, 15(5):308-318,1972.

[5] Peter J. Denning. Virtual memory. ACM Comput. Surv., 2(3):153-189, 1970.

[6] Dale H. Grit. Global lru page replacement in a multiprogrammed environment. In Robert L. Morrison, editor, Int. CMG Conference, pages 265-270. Computer Measurement Group, 1977.

[7] Donald R. Innes. Exploiting the least recently used page replacement algorithm. Softw., Pract. Exper., 7(2):271-273, 1977.

[8] Ben H. H. Juurlink. Approximating the optimal replacement algorithm. In Stamatis Vassiliadis, Jean-Luc Gaudiot, and Vincenzo Piuri, editors, Conf. Computing Frontiers, pages 313-319. ACM, 2004.

[9] Richard Y. Kain. How to evaluate page replacement algorithms. In SOSP, pages 1-5, 1975.

[10] Sami Khuri and Hsiu-Chin Hsu. Visualizing the cpu scheduler and page replacement algorithms. In SIGCSE, pages 227-231, 1999.

[11] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans. Computers, 50(12):1352-1361, 2001.

[12] John S. Liptay. Structural aspects of the system/360 model 85 ii: The cache. IBM Systems Journal, 7(1):15-21, 1968.

[13] Elizabeth J. O'Neil, Patrick E. O'Neil, and GerhardWeikum. Anoptimality proof of the lru- page replacement algorithm. J. ACM,46(1):92-112, 1999.

[14] Barton G. Prieve and Robert S. Fabry. Vmin-an optimal variable-space page replacement algorithm. Com- mun. ACM, 19(5):295-297, 1976.

[15] Gururaj S. Rao. Performance analysis of cache memories. J. ACM, 25(3):378-395, 1978.

[16] William D. Strecker. Cache memories for pdp-11 family computers. In ISCA, pages 155-158, 1976.

[17] Andrew S. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.

[18] Rollins Turner and Henry M. Levy. Segmented fifo page replacement. In SIGMETRICS, pages 48-51, 1981.