International Journal of Hybrid Information Technology
Vol.2, No.2, April, 2009

A Replacement Algorithm Based on Weighting and Ranking Cache Objects

Kaveh Samiee
Iran University of Science and Technology
Department of Electrical Engineering
Narmak, Tehran, Iran
ksamiee@ee.iust.ac.ir

Abstract

Caching is one of the major steps in system designing to reach a higher performance in operating systems, databases and World Wide Web. High performance processors need memory systems with a proper access time, but still there is a big gap between performances of processors and memory systems. Virtual memory management and hierarchical memory models play an important role in system performance. In these architectures caching replacement policy defines the enhancement factor of the memory system and could modify the efficiency of the system. Different caching policies have different effects on the system performance. Because of the highlight role of replacement policies in the systems, there have been lots of work and proposed algorithms to overcome the problem of performance gap between processor and memory. Most of these policies are the enhancement of the Least-Recently-Used (LRU) and Least-Frequently-Used (LFU) schemes. Although most of the proposed schemes could solve the defects of the LRU and LFU, but they have lots of overhead and are difficult to implement. The most profit of LRU and LFU is their simple implementation. This article proposes an adaptive replacement policy which has low overhead on system and is easy to implement. This model is named Weighting Replacement Policy (WRP) which is based on ranking of the pages in the cache according to three factors. Whenever a miss occurs, a page with the lowest rank point is selected to be substituted by the new desired page. The most advantage of this model is it's similarity to both LRU and LFU, which means it has the benefits of both (i.e. in cases like loops in which LRU fails, it will switch to LFU). Simulations show that this algorithm performs better than LRU and LFU. In addition, it performs similarly to LRU in the worst cases. The new approach can be applied as a replacement policy in virtual memory systems and web caching.

1. Introduction

Caching is an old and well-known performance enhancement technique that is widely used in computer systems. Processor's cache, paged virtual memory and web caching are some instances of using this technique. In all levels of storage hierarchy, performance of the system is related to cache mechanism. The time access for cache is at least 10 times smaller than the main memory. Increasing the cache size results in better performance, but it's very expensive and it has a very expensive technology. That's why the size of cache memory is always smaller than the main memory in every system. According to cache memory access time, fetching data from cache is a big advantage and has a significant role in system performance. We must use other techniques to make cache more efficient for systems which make use of it in their memory hierarchy. The terminology used for cache efficiency is "Hit Rate" (Hit Rate -

represents the rapport between the number of addresses accessed from cache and the total number of addresses accessed during that time). The antonym for "Hit Rate" is named "Miss Rate" which can be determined by Miss Rate = 1 – Hit Rate formulas. Cache efficiency could change by implementing different replacement policies. Cache and auxiliary memories use uniformly sized items which are called pages. When the cache is empty the first access will be always a miss. The misses may occur until the cache is filled with pages. After some cycles, when the cache is filled, this kind of misses will not occur anymore. Another kind of miss occurs when the cache is full of pages and there is an access to a page that is not in the cache, in this case replacement policy should specify conditions under which a new page will be replaced with an existing one. The main role of replacement policy is to choose a page which has the lowest probability of further usage in the cache to be replaced with the new one. Generally there are three ways of mapping memory into cache: 1) Direct Mapped in which each block from the main memory is mapped only to a unique cache block, no matter whether that block is empty or not. This model of cache organization doesn't require a replacement policy. 2) Fully Associative in which each block from the main memory can be mapped to any of cache blocks. If the cache is full then a replacement policy needs to decide which data will be invalidated from cache for making room of this new data block. 3) Set Associative in which the cache is split into many sets. The new data can be mapped only to the blocks of a certain set, determined by some bits from address field. In practice it is used mostly because of a very efficient ratio between implementation and efficiency. There may be another kind of miss in the direct map caches which is due to the same data found in many places in the cache.

An optimal replacement policy must have good performance with low overhead on the system and it must be easy to implement in hardware. It means, there is a trade-off between a replacement policy hit ratio, it's implementation difficulties and it's overhead and cost. FIFO (First In First Out) may be a good example which has very simple implementation, but gets into problem when the size of physical memory is big. The problem with FIFO is that it ignores the usage pattern of the program. Most replacement algorithms in use are based on recency and frequency features, but they fail in some applications. Other new policies perform in better ways, but most of them are difficult to implement. For instance, a recency-based policy, like LRU fails badly for loops slightly larger than memory. A frequency-based policy, like LFU, works badly when different parts of memory have different time–variant patterns. The LFU policy has several drawbacks: it requires logarithmic implementation complexity in cache size, pays almost no attention to recent history, and does not adapt well to changing access patterns since it accumulates stale pages with high frequency counts that may no longer be useful. The ARC (Adaptive Replacement Cache), combines the LRU and LFU solutions and dynamically adjusts between them. ARC like LRU is easy to implement and has low over-head on system [1-8].

Many replacement policies have been proposed in the last few years, like LRU-K algorithm, the GD-size policy [6] and GDSF algorithm, which is an enhancement of the GD-size algorithm. Most of these caching algorithms like TSP [1] and SEQ [4] are based on recency, frequency, size and cost. Most of them are suitable for web caching. LRU-K is a practical implementation of the LRU policy. The basic idea behind the LRU-K replacement policy is that it keeps track of the last K references. When the cache becomes full it replaces the page with the largest K length. However, there can be

several pages with the maximum length of K, and when this happens several different routes can be taken to decide which one to choose. In this tool it cycles through the pages to attempt to keep the same page from being replaced every time it comes across this situation. LRU-K does not have a higher hit ratio than the ideal LFU policy in most cases. However, LRU-K is more practical because it limits the number of bits used to monitor how recently a value was last used.

LRU captures only recency and not frequency, and can be easily "polluted" by a scan. A scan is a sequence of one-time use only page requests, and leads to lower performance. ARC overcomes these two downfalls by using four doubly linked lists. Lists T_1 and T_2 are what is actually in the cache at any given time, while B_1 and B_2 act as a second level. B_1 and B_2 contain the pages that have been thrown out of T_1 and T_2 respectively. The total number of pages therefore needed to implement these lists is $2*C$, where C is the number of pages in the cache. T_2 and B_2 contain only pages that have been used more than once. The lists both use LRU replacement, in which the page removed from T_2 is placed into B_2. T_1 and B_1 work similarly together, except where there is a hit in T_1 or B1 the page is moved to T_2. The part that makes this policy very adaptive is the sizes of the lists change. List T_1 size and T_2 size always add up to the total number of pages in the cache. However, if there is a hit in B_1, also known as a Phantom Hit (i.e. not real hit in cache), it increases the size of T1 by one and decreases the size of T_2 by one. In the other direction, a hit in B_2 (Phantom Hit) will increase the size of T_2 by one and decrease the size of T_1 by one. This allows the cache to adapt to have either more recency or more frequency depending on the workload [8]. CAR (*Clock with Adaptive Replacement*) works in much the same way that ARC does. However, CAR avoids the last two downfalls of LRU that ARC does not. These downfalls are: 1) the overhead of moving a page to the most recently used position on every page hit, and 2) serialized, in the fact that when moving these pages to the most recently used position the cache is locked so that it doesn't change during this transition. This then causes delays in a multi-thread environment. In reality CAR only overcomes the first downfall, the second still has a presence. CAR uses two clocks instead of lists, a clock is a circular buffer. The two circular buffers are similar in nature to T_1 and T_2, and it does contain a B_1 and B_2 as well. The main difference is that each page in T_1 and T_2 contain a reference bit. When a hit occurs this reference bit is set on. T_1 and T_2 still vary in size the same way they do in ARC (i.e. Phantom Hits cause these changes). When the cache is full the corresponding clock begins reading around itself (i.e. the buffer) and replaces the first page whose reference bit is zero. It also sets the reference bits back to zero if they are currently set to one. The clock will continue to travel around until it finds a page with reference bit equal to zero to replace [10].

The CAR policy reduces overhead compared to LRU. This cannot be seen when viewing the simulator, however the CAR policy uses a circular buffer and this removes the overhead of having to move the most-recently used position on every page hit. The following sequence however shows that it can outperform the policies used today (ARC, LRU, LRU-K). CAR does not outperform the ideal LFU in terms of hit ratio, however LFU has much more overhead because it keeps track of the number of times a value is used for an infinite number of values. See [2, 7, 8, 21] for more details.

Adaptive replacement policy is a powerful tool and can increase the system performance by making itself adaptive with memory reference behaviors. We propose a new replacement algorithm which is based on both recency, frequency and considering rate between the references of each object. In fact, it performs both LRU and LFU algorithms and gives a weight to each page according to used parameters of those algorithms. Considering reference rate can have some benefits and may become a powerful tool for specifying replacement conditions especially for applications like web caching. The basic idea of our algorithm is to rank pages based on their recency, frequency and reference rate. So, pages that are more recent and have been used frequently are ranked higher. We also consider an additional factor, reference rate, so pages with smaller reference rate will rank higher. It means that the probability of using pages with small reference rate is more than the one with bigger reference rate.

The best advantage of our algorithm is that it can behave like both LRU and LFU by replacing pages that were not recently used and pages that are accessed only once. For linear loops in which LRU fails, it can perform better, but never performs worse than LRU and LFU.

We call this algorithm *Weighting-Replacement-Policy* (WRP), which means that this replacement algorithm ranks objects in the buffer and gives a weight to each object.

This article is organized as follows. In Section 2, the origins of the idea are described. In Section 3, we propose our method and its details. In Section 4, we show implementation and simulation results to prove the advantages of our replacement policy. At the end, we discuss about future work in section 5.

2. The Origins of the Idea

Caching is a mature technique that has been widely applied in many computer science areas, two of the most important of which are Operating Systems and Databases. Currently, the World Wide Web is becoming another popular application area of caching. The optimal algorithm is one that will cause the least number of page faults. To do this, we need future knowledge of the program execution, which is not realistic in most time. The LRU Algorithm has been widely used in commercial systems as page replacement policy and it could be the first choice for those systems. It is based on the assumption that pages which have not been referenced for the longest time won't be used in near future. In other word, the reference probability of pages that have been heavily used is more than the other pages.

LFU considers the number of times that a page has been referenced and it chooses the page with lowest referenced number to replace. In fact, LFU considers recency feature, but LFU considers frequency features. LRU algorithm only keeps the time of last reference to each page and does not have the ability to distinguish between frequently used pages and infrequently referenced pages. Thus, these two algorithms could complete each other [3].

The memory system does not consist of a single cache, but a hierarchy of caches. If each cache in the hierarchy decides which block to replace in an independent manner, the performance of the whole memory hierarchy will suffer, especially if inclusion is to be maintained. For example, if the level 2 cache decides to evict an LRU block, one or more blocks at level 1, possibly not LRU, can be evicted, therefore affecting the hit rate

of level 1 and the overall performance [11]. In these cases a global strategy usually has a limited performance potential. But an optimal global strategy could help to improve the performance. By implementing a global replacement policy for all cache levels which includes both recency and frequency features, the hit ratio of each level would be in the minimum influence of the upper level.

The LRU-K algorithm has been developed to optimize LRU, The basic idea of LRU-K is to consider the time of the last K references to a page and to use such information to make page replacement decision [5].

3. Concepts of the Ranking Policy by Weighting

In this section we will use standard replacement policies terminology. We assume that the size of the blocks is equal and the replacement algorithm is used to manage a memory that holds a finite number of these blocks. A *hit* will occur when there is a reference to a block in the buffer. When we have a reference to a block not in the buffer, a *miss* will occur and referenced block must be added to buffer. When buffer is full and a miss occurs, an exciting block must be evicted to make room for a new one.

For ranking referenced pages, we consider three factors. Let L_i be the counter which shows the recency of block i in the buffer and F_i be the counter which shows the number of times that block i in buffer has been referenced. Consider the time difference $\Delta T_i = Tc_i - Tp_i$ where Tc_i is the time of last access and Tp_i is the time of penultimate access. Then the weighting value of block i can be computed as

$$W_i = \frac{L_i}{F_i \times \Delta T_i} \quad (1)$$

L_i will work like LRU counter, when the new object k is placed in the buffer then all of the above mentioned factors in weighting function must be set, and it will be followed by setting L_k to 0, F_k to 1 and ΔT_k to 1. F_k has been set to 1 because it means that object k has been used once and we assume initial value of ΔT_k equals to 1 because the time between each reference to a block would be at least one in its minimum case. In every access to buffer, if referenced block j is in the buffer then a hit is occurred and our policy will work in this way:

1) L_i will be changed to $L_i + 1$ for every $i \neq j$.
2) For $i = j$ first we put $\Delta T_i = L_i$, $F_j = F_j + 1$ and then $L_j = 0$.

But if referenced block j is not in the buffer a miss occurs and the algorithm will choose the block in buffer which the value of its weighting function is greater than the others. Searching for object with greatest weighting value will be started in the buffer from top to down. In this way, if values of some object are equal to each other, the object which is placed upper in the buffer will be chosen to evict from buffer. It means that our policy follows FIFO low in its nature. Let assume that a miss has been occurred and block k has the greatest weighting value and it must be evicted from buffer, first we change L_i to $L_i + 1$ for every $i \neq k$ and then replace new referenced block with block k and at last we must set all weighting factors of block k to their initial values.

The weighting value of blocks that are in buffer will update in every access to cache. The value of weighting function can never be greater than the value of LRU counter and when cache size is small or time between hits is greater than cash size our algorithm will work like LRU, but as we showed in simulation section our algorithm will work better than LRU and LFU in most of applications with different cache sizes.

We have only talked about how our replacement algorithm works, but we have not discussed how it can be implemented in system mechanisms. One of the important concepts in replacement policies is its overhead in the systems.

WRP needs three elements to work and will add space overhead to system: first, algorithm needs a space for counter L_i second, it needs a space for counter F_i and third it needs a counter for ΔT_i. The last and maximum space that it needs is a space for W_i, which is as weighing value for each object in the buffer. Calculating weighting function value for each object after every access to cache will cause a time overhead to system. Although we may consider the W_i as an integer number to decrease the time and memory overheads.

4. The Results of Simulation

To evaluate our replacement algorithm experimentally, we simulated our policy and compared it with other policies like LRU and LFU. The simulator program was designed to run some traces of load instructions executed for some real programs and implement different replacement policies with different cache sizes. The obtained *hit ratio* depends on the replacement algorithm, cache size and the locality of reference for cache requests. Modular design of simulator program allows easy simulation and optimization of the new algorithm.

4.1. Input traces

An address trace is simply a list of millions memory addresses produced by a real program running on a processor. These are the addresses resulting from load and store instructions in the code as it is executed. Some address traces would include both instruction fetch addresses and data (load and store) addresses, but you will be simulating only a data cache, so these traces only have data addresses. We used four traces to simulate our algorithm which were taken from the SPEC benchmarks. The mapping scheme is considered and set to *set associative* mapping. According to traces that have been used, we considered 10 cache sizes and ran the simulation program to test the performance of our proposed algorithm.

4.2. Simulation Results

We executed simulation program for case swim, with all 10 different cache sizes and compared it with LRU and LFU. As it is indicated in Figure1, our weighting replacement policy performs clearly better than LRU and LFU. Although, for some cache sizes all of three implemented algorithms have approximately the same hit ratio. The average enhancement compare with LRU for *swim* trace is about 1.2% and average enhancement for LFU is about 0.8%. Figure 1 shows the comparison and the result for this trace. In Table 1, we can see the accurate hit ratio for all cache sizes for all three schemes. The simulator computed the accurate W_i for each references, but for implementing there is no necessarily to use float value and it can be simplified to an integer number.

For the other three cases we executed simulator program in the same way with 10 cache sizes. The second trace that we used was *bzip.* First, we ran program with implementing LRU and LFU algorithms. The result showed that the maximum hit ratio for *bzip* is achieved for LFU mode with the largest cache size and it was about 73%. We expected that the result for our algorithm be at least equal to the highest value between LRU and LFU or more than it. The result showed that the maximum hit ratio for WRP is about 76%, and is 3% more than the best hit ratio that is achieved for both of LRU and LFU.

Figure 2, 3, 4, show the graph comparisons and results for *bzip, gcc* and *six pack* traces, respectively. In all the used traces the hit ratio of proposed algorithm is always better than both LRU and LFU replacement policies, and in the worse cases the result is equal to the maximum hit ratio of LRU and LFU. That is because of the considered weighting factors in WRP which never let the result become worse than maximum hit ratio of these replacement policies. In some cases the hit ratio for our replacement policy is about 5% better than the results of LFU and LFU with the same cache sizes. In some cases for small cache sizes the *hit ratio* for all schemes are equal (i.e. *gcc* and *six pack* traces, which by increasing the cache size the enhancement of the WRP slightly will appear). Table 2, 3, 4 show the hit ratio values for each considered cache size of the other three traces *bzip*, *gcc* and *sic pack* respectively, for trace driven simulation of LRU, LFU and WRP schemes.

Cache Size (# of Blocks)	LRU %	LFU %	WRP %
100	60.5683	60.5685	62.5238
200	63.7365	63.7365	63.7365
300	70.4976	70.5045	70.8542
400	74.8617	74.9678	74.9678
500	76.0234	75.9876	76.4376
600	77.7021	77.9921	78.6563
700	79.0105	79.7589	80.6491
800	80.5689	81.1175	81.2197
900	84.8996	85.0641	85.9829
1000	89.0164	89.7329	92.8481

Table 1. A Comparison between hit ratio of LRU, LFU and WRP for 10 different cache sizes. (*swim*). In this case we can see a significantly enhancement compare to both LRU and LFU for WRP in all cache sizes.

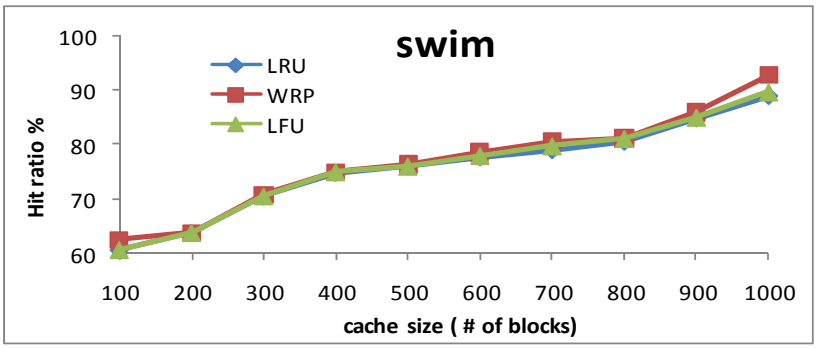

Figure1. Performance of LRU, LFU and WRP with different cache sizes for *swim*. WRP performs better than both other algorithms

Cache Size (# of Blocks)	LRU %	LFU %	WRP %
100	18.8436	18.8436	18.8551
200	19.5273	19.5273	19.5273
300	22.9873	22.9873	22.9873
400	26.9509	26.969	26.9613
500	35.619	36.7377	38.2146
600	43.1042	43.9813	45.4251
700	50.6731	51.1496	53.743
800	60.1825	61.0492	64.8525
900	66.1394	67.257	70.2583
1000	71.9765	72.7491	75.8313

Table 2. A Comparison between hit ratio of LRU, LFU and WRP for 10 different cache sizes. (*bzip*). In this case, by increasing the size of cache there will be a better enhancement in hit ratio for WRP vs. LRU and LFU.

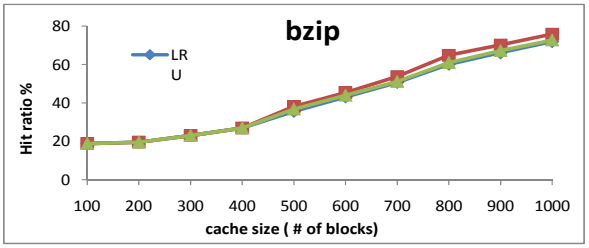

Figure2. Performance of LRU, LFU and WRP with different cache sizes for *bzip*. WRP performs better than both other algorithms

Cache Size (# of Blocks)	LRU %	LFU %	WRP %
100	13.0274	13.0274	13.0274
200	13.0992	13.0992	13.0992
300	14.6674	14.6674	14.6674
400	15.4761	15.4761	15.4761
500	22.9822	21.7643	23.2147
600	37.5728	36.966	39.6959
700	50.1495	48.6481	54.8712
800	58.7752	57.5832	61.873
900	69.4381	68.5814	72.4963
1000	71.0957	70.1239	73.8955

Table 3. A Comparison between hit ratio of LRU, LFU and WRP for 10 different cache sizes. (*gcc*). At small cache sizes there is not significant enhancement, but by increasing the cache size the enhancement will appear.

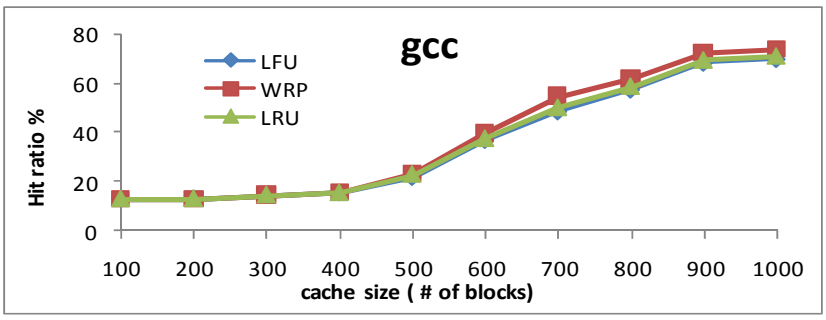

Figure3. Performance of LRU, LFU and WRP with different cache sizes for *gcc*. WRP performs better than both other algorithms

Cache Size (# of Blocks)	LRU %	LFU %	WRP %
100	8.2776	8.2776	8.2776
200	8.767	8.767	8.767
300	9.2061	9.2061	9.2061
400	11.0102	10.0102	11.0102
500	20.864	19.236	20.864
600	42.7649	39.851	42.7649
700	59.8541	56.8534	59.8541
800	62.6738	60.3293	64.1478
900	64.9531	63.7287	66.0025
1000	67.7395	66.1052	70.98

Table 3. A Comparison between hit ratio of LRU, LFU and WRP for 10 different cache sizes. (*sixpack*). As it can seen, in this case when there is no enhancement the hit ratio of WRP is equal to LRU and this value is always better than LFU's one.

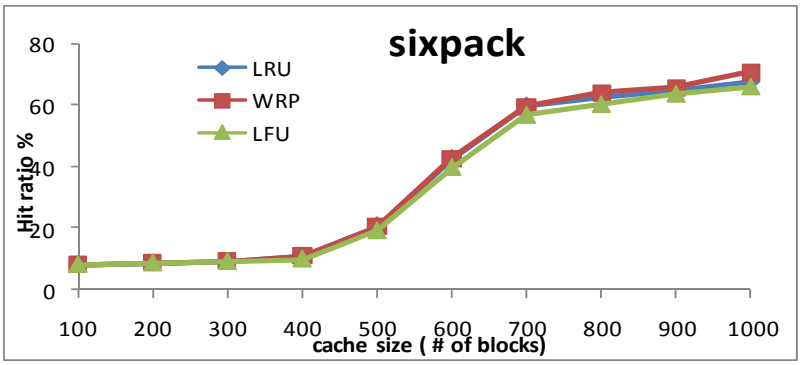

Figure4. Performance of LRU, LFU and WRP with different cache sizes for *sixpack*. WRP performs better than both other algorithms

5. Summary & Conclusions

In this article we have introduced a new caching replacement policy and the trace driven cache simulation showed that it was a modification and improvement of famous replacement policies like LRU and LFU. We attended to reference rate of each object in the buffer and claimed that the probability of re-accessing of the pages with smaller weighting value is more than the others and simulations confirmed our claim. We simulated WRP with only four traces and compared it with only two famous policies. This algorithm can be simulated with other traces and for different applications and then the overall results may be compared with the results of recently proposed replacement policies.

It is worth mentioning that other features, additional parameters and factors which describe features of objects in the buffer must be taken into account, for instance, considering the cost and size of each object in the cache that will make WRP suitable for applications like web caching. Another factor that would help to make this scheme more adaptive is to expand ΔT_i by considering the time of the previous accesses.

References

[1] Q. Yang, H. H. Zhang and H. Zhang, "Taylor Series Prediction: A Cache Replacement Policy Based on Second-Order Trend Analysis," *Proc. 34th Hawaii Conf. System Science,* 2001.

[2] S. Hosseini-khayat, "On Optimal Replacement of Nonuniform Cache Objects," *IEEE Trans. Computers*, vol. 49, no.8, Aug. 2000.

[3] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM systems J. vol. 5, no.2, pp.78-101, 1966.

[4] S. Irani, "Page Replacement with Multi-Size Pages and Applications to Web Caching," *Proc.29th Ann, ACM symp. Theory of Computing*, pp. 701-710, 1997.

[5] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "an Optimality Proof of the LRU-K page Replacement Algorithm." *J.ACM*, vol. 46, no.1, pp. 92-112, 1999.

[6] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", *proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems*, May 1997, pp. 115-12

[7] S. Jihang and X. Zhang, "LIRS: An Efficient Low Inter Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. ACM Sigmetrics Conf.*, ACM Pres, pp. 31-42, 2002.

[8] N. Megiddo and D. S. Modha, "ARC: A Self-Tunning, Low Overhead Replacement Cache," *Proc.Usenix Conf. File and Storage Technologies* (FAST 2003), Usenix, 2003, pp.115-130

[9] Y. Zhou and J. F. Philbin, "The Multi-Queue Replacement Algorithm for Second for Second-Level Buffer Caches," *Proc. Usenix Ann. Tech conf.* (Usenix 2001), Usenix, 2001, pp. 91-104.

[10] Sorav Bansal and Dharmendra S. Modha, "CAR: Clock with Adaptive Replacement." USENIX File and Storage Technologies (FAST), March 31-April 2, 2004, San Francisco, CA.

[11] Mohamed Zahran. "Cache Replacement Policy Revisited," In *Proceedings of the 6th Workshop on Duplicating Decon-structing, and Debugging,* San Diego, CA, USA, June 2007.

[12] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," Comm. ACM, vol. 29, no. 4, pp. 320–330, 1986.

[13] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 161– 203, 1985.

[14] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[15] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, 1971.

[16] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, pp. 1352–1360, 2001.

[17] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.

[18] C. Aggarwal, J. L. Wolf, and P. S. Yu. "Caching on the WorldWideWeb," In *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, pp. 94-107, 1999.

[19] Yannis Smaragdakis, Scott Kaplan, Paul Wilson, "The EELRU adaptive replacement algorithm*"* p*erformance Evaluation,* v.53 n.2, pp. 93-123, July 2003.

[20] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–143, 1999.

[21] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. VLDB Conf.*, pp. 297–306, 1994.

[22] S. Albers, S. Arora, and S. Khanna, "Page replacement for general caching problems," *Proceedings of the* 10*th Annual ACM–SIAM Symposium on Discrete Algorithms*, pp. 31–40, 1999.

Authors

Kaveh Samiee received his Bachelor's degree from the department of Electronics Engineering in Mazandaran University, Babol, Iran, in September 2005, and his Master's degree from Iran University of Science and Technology (IUST), Tehran, Iran, January 2008. His prime research interest includes digital signal and image processing, object detection and pattern recognition, computer arithmetic and performance evaluation.