Analysis and Predictability of Page Replacement Techniques towards Optimized Performance

Debabrata Swain
Dept. of CSE
SPCOE
Pune, India

Bancha Nidhi Dash
Dept. of CSE
GITA
Bhubaneswar, India

Debendra O Shamkuwar
Dept. of CSE
SPCOE
Pune, India

Debabala Swain
Dept. of CSE
CUTM
Bhubaneswar, India

ABSTRACT
Caching is a fundamental technique commonly employed to hide the latency gap between memory and the CPU by exploiting locality in memory accesses. On today's architectures a cache miss may cost several hundred CPU cycles [1]. In a two-level memory hierarchy, a cache performs faster than auxiliary storage, but is more expensive. Cost concerns thus usually limit cache size to a fraction of the auxiliary memory's size.This paper represents a comparative predictability about some of the traditional and new replacement techniques in contrast with OPTIMAL replacement technique.

General Terms
Computer Architecture, Performance Evaluation of Computer Systems

Keywords
Memory Management, Cache Performance, Replacement Policy, Hit Ratio Analysis.

1. INTRODUCTION
Page replacement is an important component of a modern operating system. When a page containing a desired datum or instruction is searched in translation look aside buffers or page tables and found missing from main memory, a page fault is said to occur. As the size of main memory is limited and is much smaller than the size of permanent storage, the role of page replacement is to identify the best page to evict from main memory as a result of a page fault and replace it by the a new page from disk that contains the requested datum or instruction. The problem is very similar to the block replacement in cache memories except that the page replacement is more critical as page transfers from disk to memory are orders of magnitudes slower than block transfers from main memory to the cache memory [25].Many page replacement algorithms are derived and tested. Some of them include First-In-First-Out (FIFO), Least Recently Used (LRU), Least Recently Used with K references (LRU-K), Random, Clock with Adaptive Replacement (CAR), Adaptive Replacement Cache (ARC**)** and at last the most efficient rather impractical Optimal algorithm. Good replacement can reduce the page fault cost resulting in higher performance, since the more page faults the operating system encounters, the more resources are wasted on paging in/out instead of doing useful work, resulting ultimately in serious thrashing problems.

In this paper, we present the relative competitive analysis for a large class of replacement policies, including LRU-K, FIFO, RANDOM, ARC, CAR and OPTIMAL. Relative competitive ratios bound the performance of one policy relative to the performance of another policy. These performance relations allow us to use cache-performance predictions.

2. BACKGROUND
Caching is a mature technique that has been widely applied in many computer science areas, Operating Systems and Databases are two most important ones. Currently, the World Wide Web is becoming another popular application area of caching. Caches are very fast but small memories that store a subset of the main memory's contents to bridge the performance difference between main memory and the processor. To reduce traffic and management overhead, the main memory is logically partitioned into a memory blocks B of size b bytes. Memory blocks are cached as a whole in cache lines of equal size. When accessing a memory block one has to determine whether the memory block is stored in the cache (cache hit) or not (cache miss). To enable an efficient look-up, each memory block can be stored in a small number of cache blocks. For this purpose, caches are partitioned into equally-sized cache sets. The size of a cache set is called the associativity k of the cache. Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called replacement policy must decide which memory block to replace upon a cache miss. To facilitate useful replacement decisions a number of status bits is maintained that store information about previous accesses. We only consider replacement policies that have independent status bits per cache set. Almost all known policies comply with this [1].

3. REPLACEMENT ALGORITHMS
Let us briefly explain the fundamental and commonly used replacement policies under investigation in the course of the paper:

The **LRU** (Least Recently Used) 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. LRU is used in the INTEL PENTIUM I and the MIPS 24K/34K.

The algorithm LRU has many disadvantages [8, 11]:
1. On every hit to a cache page it must be moved to the most recently used (MRU) position. In an asynchronous computing environment where multiple threads may be trying to move pages to the MRU position, the MRU position is protected by

a lock to ensure consistency and correctness. This lock typically leads to a great amount of contention, since all cache hits are serialized behind this lock. Such contention is often unacceptable in high performance and high throughput environments such as virtual memory, databases, file systems, and storage controllers.

2. In a virtual memory setting, the overhead of moving a page to the MRU position–on every page hit–is unacceptable [24].

3. While LRU captures the "recency" features of a workload, it does not capture and exploit the "frequency" features of a workload [23, p. 282]. More generally, if some pages are often requested, but the temporal distance between consecutive requests is larger than the cache size, then LRU cannot take advantage of such pages with "long-term utility".

4. LRU can be easily polluted by a scan, that is, by a sequence of one-time use only page requests leading to lower performance.

LRU-K (Least Recently Used with K references): The basic idea of LRU-K is to keep track of the times of the last K references to popular database pages, using this information to statistically y estimate the interarrival times of references on a page by page basis [26].

The LRU-K algorithm is based on the following data structures: l HIST(p) denotes the history control block of page p; it contains the times of the K most recent references to page p, discounting correlated references: HIST(p,l) denotes the time of last reference, HIST(p,2) the time of the second to the last reference, etc.l LAST(p) denotes the time of the most recent reference to page p, regardless of whether this is a correlated reference or not. These two data structures are maintained for all pages with a Backward K-distance that is smaller than the Retained Information Period. An asynchronous demon process should purge history control blocks that are no longer justified under the retained information criterion. [6, 26]

Random Replacement (RR): It is used to randomly select a candidate item and discard it to make space when necessary. This algorithm does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors [27]. It admits efficient stochastic simulation.

Random replacement algorithm replaces a random page in memory. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice. OS/390 uses global LRU approximation and falls back to random replacement when LRU performance degenerates, and the Intel i860 processor used a random replacement policy.

LFU (Least Frequently Used) is a frequency-based policy, in which the page with low frequency will be replaced first. It works badly because 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 [1-8].

In **FIFO** (First-In-First-Out) the operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list 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 [1]. FIFO is used in the INTEL XSCALE, ARM9 and ARM11.

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 systems [9].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 T1 and T2 are what is actually in the cache at any given time, while B1 and B2 act as a second level. B1 and B2 contain the pages that have been thrown out of T1 and T2 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. T2 and B2 contain only pages that have been used more than once. The lists both use LRU replacement, in which the page removed from T2 is placed into B2. T1 and B1 work similarly together, except where there is a hit in T1 or B1 the page is moved to T2. The part that makes this policy very adaptive is the sizes of the lists change. List T1 size and T2 size always add up to the total number of pages in the cache. However, if there is a hit in B1, 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 T2 by one. In the other direction, a hit in B2 (Phantom Hit) will increase the size of T2 by one and decrease the size of T1 by one. This allows the cache to adapt to have either more recency or more frequency depending on the workload [9].
The **CAR** (Clock with Adaptive Replacement) combines the LRU and LFU solutions and dynamically adjusts between them. CAR like LRU is easy to implement and has low over-head on system.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 T1 and T2, and it contains a B1 and B2 as well. The main difference is that each page in T1 and T2 contains a reference bit. When a hit occurs this reference bit is set on. T1 and T2 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 [11].

The **OPTIMAL** page replacement algorithm is easy to describe. When memory is full, you always evict a page that will be unreferenced for the longest time [28]. This scheme, of course, is possible to implement only in the second identical run, by recording page usage on the first run. But generally the operating system does not know which pages will be used, especially in applications receiving external input. The content and the exact time of the input may greatly change the order and timing in which the pages are accessed. But nevertheless it gives us a reference point for comparing practical page replacement algorithms. This algorithm is often called **OPT** or **MIN**.

Table-1 Qualitative Analysis of different Cache Techniques

ALGORITHM	PERFORMANCE CHARACTERISTICS	OVERHEAD COST
LRU, LRU-K	Easy to Implement	High
LFU	Easy to Implement	High
FIFO	May throw important pages	Low
RANDOM	Efficient for stochastic simulations	Low
CAR	Complex	Fair
ARC	Complex	Low
OPTIMAL	Not implementable, can be used as a benchmark	Low

4. PERFORMANCE ANALYSIS

To evaluate our replacement algorithm experimentally, we simulated our policy and compared it with other policies like LRU-K, FIFO, RANDOM, ARC and CAR and finally OPTIMAL. The simulator program was designed to run some real time 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

Our address trace is simply a list of thousands of memory addresses produced by a real program running on a processor. Generally address traces would include both instruction fetch and data fetch (load and store), but we are simulating only a data cache, so these traces only have data addresses. The mapping scheme is considered and set to *set associative* mapping. According to traces that have been used, we considered 8 cache sizes and ran the simulation program to test the performance of different replacement policies like LRU-K, FIFO, RANDOM, ARC and CAR and finally OPTIMAL.

The simulator program is 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.

4.2 Simulation Results

We executed simulation program for all 8 different cache sizes and compared it with LRU-K, FIFO, RANDOM, ARC, CAR and OPTIMAL algorithms which is taken as a benchmark and all other are predicted relative to this. As it is indicated in Table 2, we can see the accurate hit ratio of all tested policies. Hit Ration can be calculated as follows:

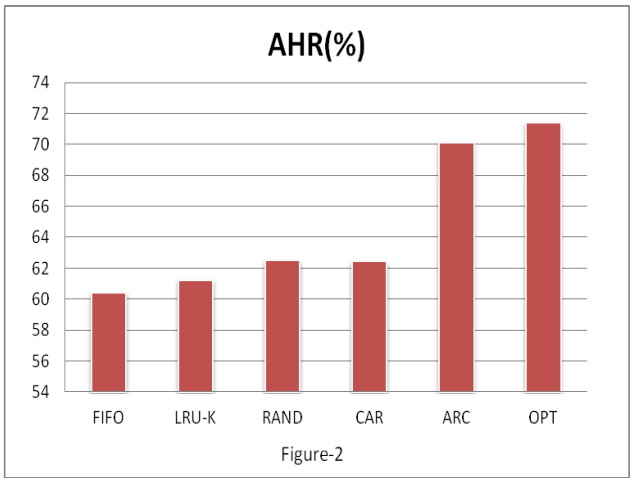
$$\text{Hit Ratio} = \frac{\text{Total No of Hit Counts}}{\text{Total No of Reference Counts}}$$

The average h it ratio of **LRU-K** is **61.22%**, **FIFO** is **60.43, CAR** is **62.42%**, **ARC** is **70.13%** and finally **OPTIMAL** is 71.24.So in average the **ARC** policy works much parallel to **OPTIMAL** and better than all other algorithms. If we will consider the individual algorithm then the result shows that the **maximum hit ratio** is **75.42%** from **CAR** is attained for the cache size of **210** and also the **minimum hit ratio** is **40.24%** at cache size **30** for **CAR** algorithm**.**

Here we have simulated the **LRU-K** algorithm with last 100 references (**k=100**) and it is concluded that it works much better than LRU algorithm. The result shows that when the cache size relatively increases then hit ratio grows linearly. It can be easily verified that both the hybrid algorithms CAR and ARC performs better than the primitive algorithms. Even if we consider the average hit ratio performance of the algorithms then it shows that **OPTIMAL** is having the **AHR** (Average Hit Ratio) of **71.25%** and **ARC** is very much nearer to this with **70.13%** .**FIFO** is having the lowest AHR of **60.43%.** It can be easily identified from figure-2.

Table 2. A Comparison between hit ratios of different algorithms for 8 different frame sizes.

Cache Size	FIFO	LRU-K	RANDOM	CAR	ARC	OPTIMAL
30	40.93	41.82	41.33	40.24	60.36	57.09
60	49.26	48.86	50.05	49.65	69.18	66.2
90	57.48	56.69	58.67	59.27	69.97	71.46
120	62.14	64.62	66.5	66.2	71.75	74.43
150	66.3	67.29	71.06	70.96	72.94	76.51
180	72.84	73.84	74.03	75.22	73.14	76.51
210	74.03	75.42	75.92	75.42	73.54	76.51



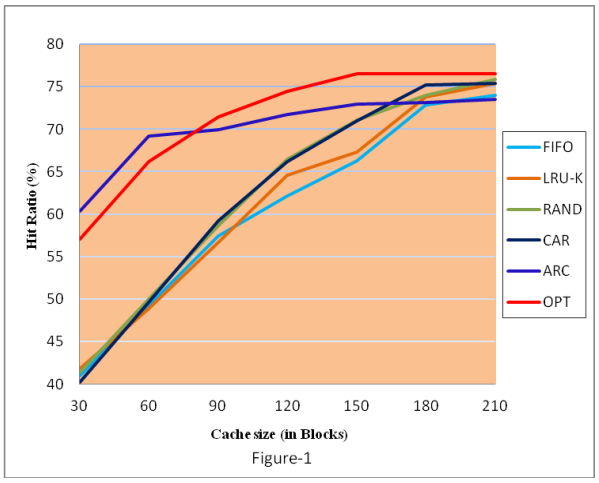


Figure 1: Performance Analysis of different techniques using Hit Ratio.

5. CONCLUSION

In this paper we have discussed various famous cache replacement policies like LRU-K, FIFO, RANDOM, ARC, CAR and OPTIMAL, then simulated and compared them to evaluate their efficiency. It makes easier to choose a specific policy for a specific set of memory reference. It also explains the variant characteristics of different algorithms, which helps us to characterize their behavior and development of new cache techniques in future development.

We have developed the tool for data cache access still preparing for instruction cache. These algorithms can also be simulated with other benchmark traces with different applications and then the overall results can be analyzed. We also expect from the readers for the development of some new hybrid algorithms like CAR and ARC which can perform better than the traditional algorithms.

6. REFERENCES

[1] Relative Competitive Analysis of Cache Replacement Policies _Jan Reineke Daniel Grund, LCTES'08, June 12–13, 2008, Tucson, Arizona, USA. Copyrightc 2008 ACM

[2]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.

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

[4] Debabala Swain, Bijay K Paikray, Debabrata Swain,"AWRP: Adaptive Weight Ranking Policy for Improving Cache Performance", Journal of Computing, vol-3, Issue-2, February 2011.

[5]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. [6] 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.

[7] 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. 115-122

[8] 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.

[9] 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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21]A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 1997.

[22] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[23]www.ecs.umass.edu/ece/koren/architecture, Computer Architecture Educational Tools.

[24] Kaveh Samiee, "WRP: Weighting Replacement Policy to Improve Cache Performance", *International Journal of Hybrid Information Technology,Vol.2,No.2, April, 2009*.

[25]Development of a Virtual Memory Simulator to Analyze the Goodness of Page Replacement Algorithms Fadi N. , Sibai, Maria Ma, David A. Lill

[26] The LRU-K Page Replacement Algorithm For Database Disk Buffering Elizabeth J. O'Neil 1, Patrick E. O'Neill, Gerhard Weikum2 SIGMOD 15193 AVaahin~ton, DC,USA @1993ACM.

[27]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexr/index.html

[28] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal, Volume 5, Issue 2, pp. 78–101 (1966).