An Improved LRU Algorithm for Replacement of Objects from Cache

Nicoleta Liviana Tudor

Department of Computer Science

Petroleum-Gas University of Ploiesti,

39 Bucuresti Avenue, 100680, Ploiesti, Romania

tudorl@upg-ploiesti.ro

Abstract: This work describes an improved LRU algorithm, called LRU-H, which uses a heuristic function for an optimum determination of the object to be removed from cache. Comparing to the use of LRU algorithm, algorithm LRU-H enables to obtain a significant decrease of the time needed to extract data from a database when using an object cache extraction and provides a hit ratio about 14.37% higher than LRU algorithm, 10.84% higher than LRFU algorithm and 7.37% higher than ARC algorithm.

Keywords: cache, LRU-H algorithm, optimization, heuristic function, hash function, hit ratio.

Nicoleta Liviana TUDOR is a PhD student at the Petroleum-Gas University of Ploiesti, Control Engineering and Computers Department. Now she is a lecturer in the Computer Science Department at Petroleum-Gas University of Ploiesti, Romania. Her research interests include: middle-tier business objects, XML web services, data processing and relational databases, contributing with 1 book and 12 papers presented at various international symposia and published in Romanian journals.

1. Introduction

Information storage systems use two types of memory: a cache and an auxiliary storage. Cache offers support for memory management in database applications, web servers, file systems, processors, operating systems; it is faster than an auxiliary storage, but is also more expensive. Both memories handle uniformly sized items called pages. Requests for pages are first directed to the cache and, if the page is not in the cache, then to the auxiliary memory. In the latter case, a copy of the page is saved into the cache. If the cache is full, one of the pages in the cache must be eliminated. A replacement policy determines which page is evicted. LRU (Least Recently Used) is the policy of choice in cache management. LRU algorithm replaces the least recently used page from the cache.

This paper describes a method of optimization of object selection within cache memory, in relational database applications. It shows an LRU-H algorithm – the LRU improved algorithm, based on a heuristic function which finds the object to be replaced from cache. **The heuristic function depends on one parameter and minimizes the sum among the number of references and the value of a Hash function, which is associated with the object. As a result, the heuristic function relocates this object to the last place within the list** LRU-H. The time of the last reference of the object in cache and the number of the effective stored objects in cache shall be considered for *choosing* the Hash function.

The paper is organized as follows:

- the section *Prior work* presents the improved alternatives of LRU algorithm
- the section *Algorithm LRU-H (Least Recently Used - Heuristic)* shows a different improved LRU algorithm based on utilization of a heuristic function and the mathematical expression of LRU-H algorithm
- the section *Implementation of a cache of objects in applications with relational databases - Case study* describes the implementation of a cache of objects and a case study – development of a JAVA application for monitoring of processes, services and drivers running on a computer.
- the section *Efficiency of objects cache* presents a comparative analysis of average values of parameters hit ratio, the size of cache and suggests the effectiveness of implementation of objects cache, using the LRU-H algorithm.

2. Prior Work

There are already known many improved alternatives of LRU algorithm i.e.:

LRU–2 [7] constitutes a significant improvement of the LRU algorithm; it memorizes for each cache page

the times of its two most recent changes. LRU-2 has a big hit ratio, but it has one disadvantage: it has logarithmic complexity because it uses a priority queue and it depends on the parameter CIP (Correlated Information Period).

2Q uses a better method than LRU-2, with constant complexity [3] and it uses a simple LRU instead of a priority queue.

LIRS (Low Inter-reference Recency Set) [2] uses a variable size LRU stack, where it selects top pages depending upon two parameters which influence its performance.

FBR (Frequency-based replacement) [9] maintains an LRU list with three sections changing pages between them. The drawbacks of FBR are its need to modify the reference counts periodically and its parameters.

LRFU (Least Recently/Frequently Used) [5] combines LRU and LFU. It assigns a value $C(x) = 0$ to every page x and, depending on a parameter $\lambda > 0$, after t time units, updates $C(x) = 1 + 2^{-\lambda} C(x)$, if x is referenced and $C(x) = 2^{-\lambda} C(x)$ otherwise. LRFU replaces the page with the least C(x) value. If λ tends to 0, then C(x) tends to the number of changes of x and LRFU tends to LFU. If λ tends to 1, then LRFU becomes LRU. The performance depends on λ. The complexity of LRFU alternates between constant and logarithmic. For small λ, LRFU can be slower than LRU.

MQ (Multi-queue replacement policy) [10] uses m queues, where the i-th queue ($0 <= i <= m-1$) contains pages that have been seen at least 2^i times, but no more than $2^{i+1} - 1$ times recently. The page frequency is incremented, a page is placed at the MRU position of the appropriate queue, and its *expireTime* is set to *currentTime* + t, where t is parameter computed from the distribution of temporal distances between consecutive accesses. MQ needs to check time of LRU pages for m queues on every request.

ARC (Adaptive Replacement Cache) [6] maintains two LRU lists of pages, L_1 and L_2 and the parameter p, $0 <= p <= c$. If the cache can hold c pages, L_1 contains p pages, which have been seen only once recently and L_2 contains c-p pages which have been seen at least twice recently. Initially, $L_1 = L_2 = \varphi$. If a requested page is in the cache, it is moved to the top of L_2, otherwise, it is placed at the top of L_1. L_1 is partitioned into T_1 (containing the most recent pages in L_1) and B_1 (containing the least recent pages in L_1), and L_2 is partitioned into T_2 and B_2. ARC delivers performance comparable to LRU, LRU-2, LIRS, 2Q [6], for example, at 16 Mb cache, LRU has a hit ratio of 4.24% and ARC achieves a hit ratio of 23.82%.

The section below shows a different improved LRU algorithm based on a heuristic function.

3. Algorithm LRU-H (Least Recently Used - Heuristic)

Algorithm LRU may be improved by implementation of a heuristic function that shall optimize the selection of object in cache, when it is necessary to remove an element. I have defined a heuristic function that determines an optimal choice of element LRU-H to be removed in case of full cache.

Heuristic function determines within cache the object having parameter p=0 and that minimizes the expression consisting of **the sum among the number of references and the value of a Hash function, which is associated with the object. As a result, the heuristic function relocates this object to the last place within the list** LRU-H.

To set up the p parameter, I will take into consideration the following cases:

Case 1: new element entry in cache

p ← 1 // p = parameter assigned to object

Case 2: the rest of Hash table elements

if p <>0

p ← p -1.

We suppose a minimum cache size of 3.

Mathematical expression of LRU-H algorithm:

To define the heuristic function the following shall be considered:

a domain $D = ((D_1, D_2, \ldots, D_n)$, D_i – range of values, i = 1, … , n, for $n \in N^*$

a relation $R(c_1, c_2, \ldots, c_n)$, c_i attribute, $c_i \in D_i$, $i = 1, \ldots, n$, c_1 - primary key of relation R

a cache object C - the set of tuples obtained after an interrogation upon R relation:

$$C = \{ (t_1, t_2, \ldots, t_n) / t_i \in D_i, i = 1, \ldots n\}$$

a Hash function defined as follows:

$$h: D_1 \rightarrow N^*, k \rightarrow h[k]$$

For each tuple $t = (t_1, t_2, \ldots, t_n) \in C$ the following shall be defined:

- t_{n+1} - a counter equal to the number of references of tuple
- p parameter, $p \in N^*$, p assigned to t:

Case 1: at addition of one tuple t* to the set C: p = 1

Case 2: for $\forall t \in C - \{t^*\}$, p = p -1 (if p <>0).

So, it is noticing:

$C' = \{ (t_1, t_2, \ldots, t_n, t_{n+1}, p) / (t_1, t_2, \ldots, t_n) \in C, t_{n+1}, p \in N^*\}$. Set C' populates cache with objects.

Further we introduce LRU-H notation.

Definition 1: LRU-H algorithm for replacement of objects from cache is LRU algorithm using a heuristic function defined as follows:

$$f : C' \rightarrow N^*, \ f(t'_i) = \min(t'_{i, n+1} + h(t'_{i, 1})),$$

$(\forall)\ ti' = (t_1, t_2, \ldots, t_n, t_{n+1}, p) \in C' - \{ t'_1\}$, $t'_{i, n+1} = t_{n+1}$, $i \in \{2, \ldots, k\}$, $k = |C'|$, $t'_{i,p} = 0$, $t'_{i, 1} \in D_1$ and h Hash function,

$h: D_1 \rightarrow N^*$, $h(t'_{i, 1}) = (\text{current\_time} - \text{last\_reference\_time}(t'_{i, 1})) \bmod k$, $k = |C'|$.

The algorithm LRU-H will work as follows:

- the entry of an element in cache C' is made at the beginning of the list, as the most recently used object
- when cache C' contains the maximum number of tuples and an element must be removed, the heuristic function f is used as it follows:
  - the tuple that minimizes the sum among the number of references and the value of a Hash function, which is associated with the object from set
  - $C' - \{ t'_1\}$ it is identified, where $C' = \{ t'_1, t'_2, \ldots, t'_{i-1}, t'_i, t'_{i+1}, \ldots. \ t'_k \}$, with $t'_{i,p} = 0$. Let it be t'_i, $i \in \{2, \ldots, k\}$, $k = |C'|$.
  - the position of element t'_i, is altered moving it to the end of the list LRU-H becoming the last one
- the least recently used element located at the end of the list LRU-H is deleted

The figure 1 shows the diagram of heuristic function:

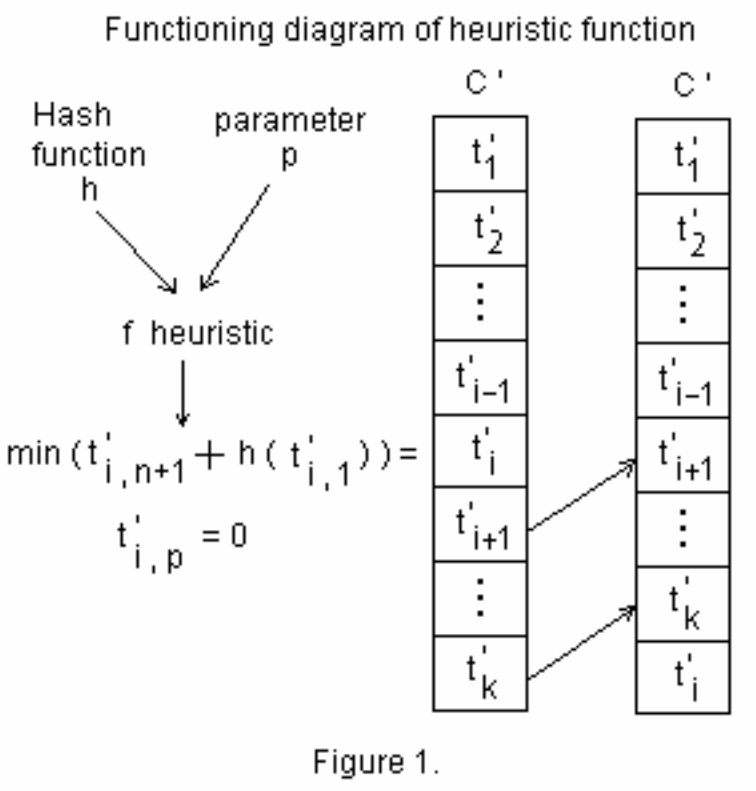

Figure 1.

Description of LRU-H algorithm in the pseudo-code language:

```
Procedure LRU-H (object)
        if [object is not in the cache]
                if [object is on the disk]
                        call add_cache( object)
                        return object
                endif
                else
                     first ← object; return object
        endif
        end
        Procedure add_cache(object)
        if cache_size = 0
                first ← last ← object
            else
                if cache_size >= max_cache_size
                     call Heuristic()
                     remove last from the cache
                endif
                first ← object
        endif
        update parameter p for all the elements from cache:
                if p <>0
                      p ← p -1
                endif
        parameter p (object) ← 1
        define the time of the last reference for object
        put object in cache (hash table)
        end
        Procedure Heuristic()
          link ← [ object that minimizes the sum between number of references of
                    object and hash function value, with parameter p = 0]
          move link to the last position
        end
```

4. Object Cache Implementation in Applications with Relational Databases – Case Study

This section describes the implementation of a cache of objects and a case study – development of a JAVA application for monitoring of processes, services and drivers running on a computer. For process administration the application uses relational database SQL Server. For the optimization of the access of data of relational database it is used a cache for temporary storage of table consisting of frequently referenced and result of queries objects.

My personal contribution consists in establishment of a module for cache administration based on LRU-H algorithm and comparative analyze and interpretation of results of algorithms for validation of suggested improvements. The implementation of a cache of objects means issuance and administration of cache of stored objects and setting the strategies for selection of the elements to be removed to free space within cache.

A cache of objects of database shall be established [1, 4] as follows:

- Objects cache may be a Hash table, with elements of a linked list
- When a client application asks for an object this will be searched within cache
- If the object is found within cache, it will be brought on the first position in the Hash table in order to be delivered to the client application
- If the object is not found within cache it will be searched on the disk, on the database server and shall be added to the cache to be used. In case of a full cache it will be used replacement algorithms for removal of one element of Hash table and release of the associate memory space. The figure 2 shows the functioning of cache:

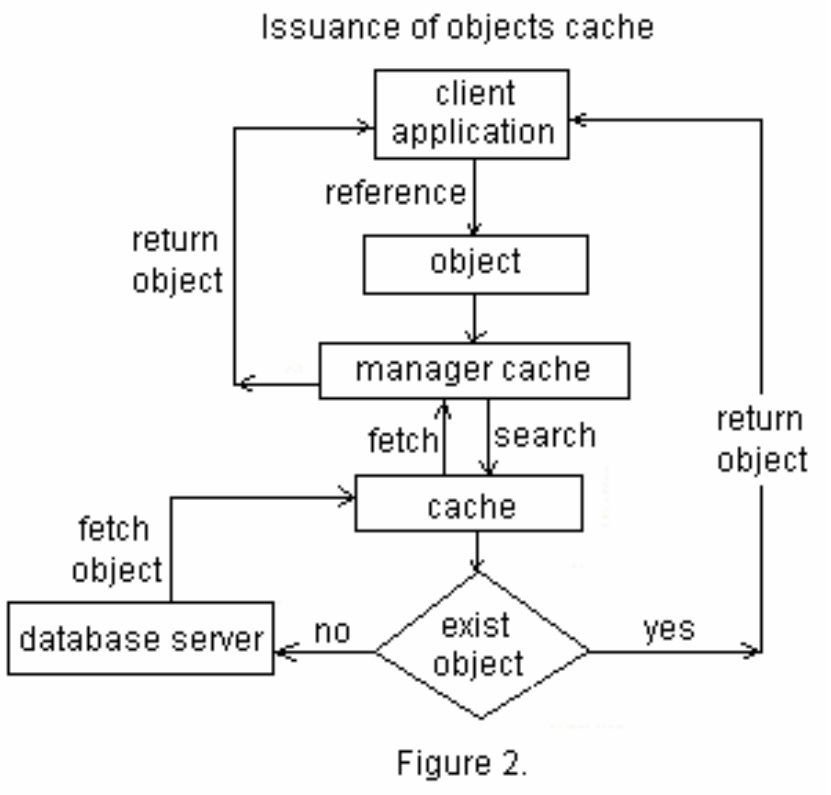

Figure 2.

LRU-H algorithm is running as cache administrator module and algorithmically it may be described as:

```
search object within cache
if object is not found within cache
        search object on the disk (on the database server)
        if object is found on the disk
```

```
                    if cache is full
                                use heuristic function
                                remove the last element
                    endif
                    update parameter p for all the elements within cache
                    define the time of the last reference for object
                    add object to cache on the first place
            endif
else
    object becomes first in cache
endif
return object
```

5. Efficiency of Cache Objects

The effectiveness of cache may be statistically determined [8], by calculating a hit ratio, which depends upon implementation of model of objects cache administration and means the percent of objects interrogations that the cache administrator is obtaining to. For example the hit ratio H means the number of accesses finding data within cache from 100 accesses. Miss ratio M – failure percent is calculated as:

$$M = 1 - H$$

When hit time (Th) is the time of reading in cache and Tm (miss time) is the time lost in case of failure, then the average access time at cache memory will be calculated bellow:

$$T = Th * H + Tm * M$$

The Miss time (Tm) equals the reading time from the slow memory (Ts) addition the cache reading time. The cache will be efficient if $T<Ts$.

The program execution has been checked using 190 tests applied to the LRU, LRU-H, LRFU and ARC algorithms for various size of cache. The results of tests enabled to determine the average values of parameter H.

The comparative analysis of average values of parameter H, average number of objects found in cache and the size of cache suggests the effectiveness of implementation of objects cache, using the LRU-H algorithm due to:

hit ratio for LRU-H algorithm is higher than for LRU, LRFU and ARC algorithms (table 1): LRU-H provides a hit ratio about 14.37% higher than LRU algorithm, 10.84% higher than LRFU algorithm and 7.37% higher than ARC algorithm.

LRU-H algorithm finds more objects within cache than LRU, LRFU and ARC algorithms (table 2).

Table 1. Performing characteristics of algorithms LRU, LRU-H, LRFU, ARC

cache size (number of objects)	hit ratio (H - %)			
	LRU	LRU-H	LRFU	ARC
10	14.79	30.31	17.77	22.18
13	15.77	30.49	19.41	23.1
15	20.19	32.42	23.82	28.16
17	20.97	34.91	24.28	28.48
20	22.85	38.3	26.97	29.51

Table 2. Average number of objects found in cache for LRU, LRU-H, LRFU, ARC

cache size (number of objects)	LRU	LRU-H	LRFU	ARC
10	4.8	7.33	5.67	6.2
13	4.88	9.25	5.75	6.88
15	5.3	7.7	6.1	7.1
17	4.57	9.43	5.14	6.43
20	4.6	9.4	6.0	7.4

Figure 3 shows the hit rates of the LRU-H policy as a function of the cache size (number of elements) for the SQL Server database used in the case study presented above. The hit rates are compared with those of the LRU, LRFU and ARC algorithms.

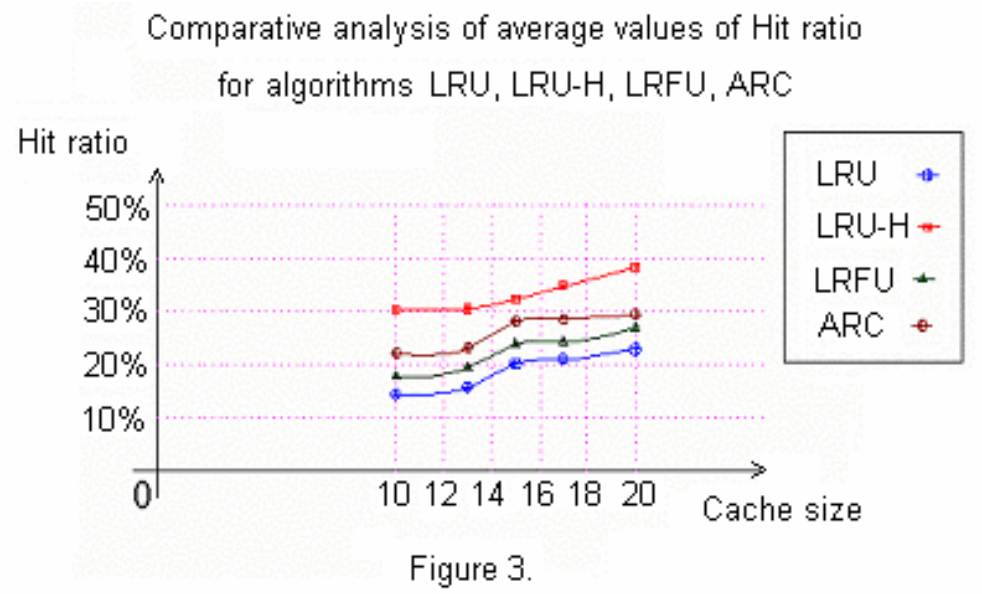

Figure 3.

The results in the figure show that the LRU-H algorithm performs more competitively with other algorithms regardless of the cache size.

6. Conclusions

This paper relates a comparative analysis of improved forms of LRU algorithm used as method of replacement of objects within cache and *emphasizes* the following personal contributions:

- Issuance of the LRU-H algorithm for optimization of objects selection within cache. LRU-H uses a heuristic function which determines the object to be removed from cache and **minimizes the sum among the number of references and the value of a Hash function, which is associated with the object. As a result, the heuristic function relocates this object to the last place within the list** LRU-H.
- Choosing an optimum Hash function for LRU-H algorithm, depending on the time of the last object reference in cache and the number of objects really stored within cache
- Creating a performing Cache Administrator module based on LRU-H algorithm.
- Thereby, using a cache based on a LRU-H algorithm, a significant decrease of data extraction time from database tables has been found.

REFERENCES

1. GRANT, M., **Cache Management**, Jupitermedia Corp., 2004.
2. JIANG, S., ZHANG, X., **LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance**, Proc. ACM SIGMETRICS Conference, 2002.
3. JOHNSON, T., SHASHA, D., **2Q: A low overhead high performance buffer management replacement algorithm**, Proc. VLDB Conference, 1994.
4. LANDGRAVE, T., **Effectively use the Cache object in ASP.NET designs**, CNET Networks, 2004.
5. LEE, D., CHOI, J., KIM, J. H., NOH, S. H., MIN, S. L., CHO, Y, KIM, C. S., **LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies**, IEEE Trans. Computers, vol. 50, No. 12, 2001.
6. MEGIDDO, N., MODHA, D. S., **A Simple Adaptive Cache Algorithm Outperforms LRU**, IBM Research Report, Computer Science, 2003.
7. O'NEIL, E. J., O'NEIL, P. E., WEIKUM, G., **An optimality proof of the LRU-k page replacement algorithm**, ACM, vol. 46, No. 1, 1999.
8. OTOO, E., OLKEN, F., SHOSHANI, A., **Disk Cache Replacement Algorithm for Storage Resource Managers in Data Grids**, IEEE Trans. on Software Eng., 2002.
9. ROBINSON, J. T., DEVARAKONDA, M. V., **Data cache management using frequency-based replacement,** Proc. ACM SIGMETRICS Conference, 1990.
10. ZHOU, Y., PHILBIN, J. F., **The multi-queue replacement algorithm for second level buffer caches**, Proc. USENIX Annual Tech. Conference, Boston, MA, 2001.