Weighting Cache Replace Algorithm for Storage System

Yihui Luo[1 2] Changsheng Xie[2] Chengfeng Zhang[2]

[1]School of mathematics and Computer Science, Hubei University, Wuhan 430062, P.R. China
[2]National Storage System Laboratory, School of Computer Science, Huazhong University of Science and Technology, Wuhan 430074, P.R. China

Abstract

Usual cache replace algorithms are based on the assumption that the high cache hit ratio can bring lower average access time, which is right as storage system has the same device access time. By analyzing the performance of storage system, a conclusion is drawn that storage system can get lower average access time only when the cache hit ratios of some objects with long device access time are higher. Based on this, some weighting cache replace algorithms such as Weighting LFU (WLFU), Weighting LRU (WLRU) and Weighting LFRU (WLFRU) are proposed. These algorithms are designed to minimize average access time and algorithm overhead. Experiment proved that WLFU and WLRU had better performance than usual algorithms such as LRU, and WLFRU had the best performance.

Keywords: Cache replace algorithm, Average access time, Algorithm overhead, Weighting value

1. Introduction

With processor speed increasing dramatically over the last few years and main memory density doubling every two years, I/O appetite continues to grow, especially with the development of applications such as multimedia and network, which place an ever-increasing demand on the storage subsystem. Although the storage device I/O speeds have increased greatly, it cannot satisfy the demand of computer. As a result, storage systems become the bottleneck of computer system and the performance of computer system is not the optimization. There are many methods to improve the performance of storage system and one of the most effective methods is to place a cache in storage system. The cache devices are usually fast storage devices but have less capacity, so the suitable replace algorithms must be used in cache to improve the performance of storage system.

There are many cache replace algorithms such as FIFO, LFU, LRU, and some varieties of them. All of them improve the cache performance by increasing the cache hit ratio based on the locality of data reference. Must the higher cache hit ratio bring the better I/O performance in the storage system? If the speeds of storage devices are the same in storage system the conclusion is right. However, storage devices usually have different I/O speeds in storage system, at the case the conclusion is no more right, so the usual algorithms are not the optimization algorithms for the storage system. Based on this standpoint, we propose a new cache replace algorithm named Weighting Least Frequently/Recently Used (WLFRU) algorithm, which is designed not to increase the cache hit ratio but to minimize the average access time of the storage system. In order to do so, WLFRU is designed to increase the cache hit ratios of objects with longer device access time, which considers not only I/O locality but the difference of device access time.

The rest of this paper is organized as the following: after analyzing the related works, we first analyze the I/O performance of storage system with cache, through which we prove that the higher cache hit ratios doesn't result to higher I/O speeds, at the same time, we can get a conclusion that the higher I/O speeds are obtained only when it is higher to the cache hit ratios of objects with longer device access time. Based on the conclusion, we then give the Weighting Least Frequently Used algorithm (WLFU), Weighting Least Recently Used algorithm (WLRU) and WLFRU. Afterwards, we give the experiment to prove our cache algorithm right. At last, we give a conclusion and the future work.

2. Related Works

There are many researches about cache replace algorithm. The algorithm LRU always replaces the least recently used objects. Various approximations and improvements to LRU abound, see, for example, enhanced clock algorithm [1]. If the workload or the

request stream is drawn from a LRU Stack Depth Distribution (SDD), LRU is the optimal policy. LRU has several advantages, for example, it is simple to implement and responds well to changes in the underlying SDD model. However, while the SDD model captures "recency", it does not capture "frequency". The algorithm LFU replaces the least frequently used objects. A relatively recent algorithm LRU-2 [2] approximates LFU, which remembers the last two times for each object, when it is requested, and to replace the object with the least recent penultimate reference. Algorithms, which consider both recency and frequency, are Frequency-based replacement (FBR) [4], Least Recently/ Frequently Used (LRFU) [5], multi-queue replacement (MQ) [6]. Based on those, an algorithm named Adaptive Replacement Cache (ARC) was proposed in [7]. The basic idea behind ARC is to maintain two LRU lists of objects. One list, say L_1, contains objects that have been seen only once "recently", while the other list, say L_2, contains objects that have been seen at least twice "recently". L_1 is thought as capturing "recency" while L_2 capturing "frequency". Although ARC captures both frequency and recency, it doesn't consider the access cost of objects. Additionally, the size of L1 is resized very frequently, which may increase the algorithm overhead.

Ekow Otoo etc gave a replace algorithm for storage source manager in data grids [4], which defined a utility function for each object to express its use status. This function relate to reference frequency, object size and device access speed, but it has several drawbacks: it pays almost no attention to recent history, and does not adapt well to changing access patterns since it accumulates stale objects with high frequency counts that may no longer be useful, moreover, it requires logarithmic implementation complexity in cache size. Ulrich Hahn etc offered a replace algorithm called ObjectLRU, which take into account influence to replace algorithm by various object properties [8]. This uses a weighting function to evaluate combinations of object properties, which provides a more flexible approach. But its weight values are difficult to select. Furthermore, the cost of algorithm realization is not low.

3. The Performance Formula of Storage System

The basal goal is to improve the I/O performance using cache in storage system. So we first calculate performance gain of storage system with cache after describing the model of hierarchical storage system. With the performance gain, we can get a few of conclusions that is the base of designing cache replace algorithm in order to optimize its I/O performance.

3.1. The Model of Hierarchical Storage System

The hierarchical storage system is described as figure 1 that consists of cache and storage devices. The cache is a small storage device with high speed, which maybe the server memory or its local disk. The storage devices include different devices such as disk, CDROM, tape or other storage devices, which may have different access speeds. The links between cache and devices may be bus or network, and their communication speeds maybe different. If it is network, its speed may vary as the network load varies. In order to simplify calculation, we suppose that the communication time is contained in access time of devices and the speeds don't vary with the network load varies. The access data maybe data blocks or files in the storage system and their sizes maybe the same or different, so we call the access data as data object, which means that their sizes are different.

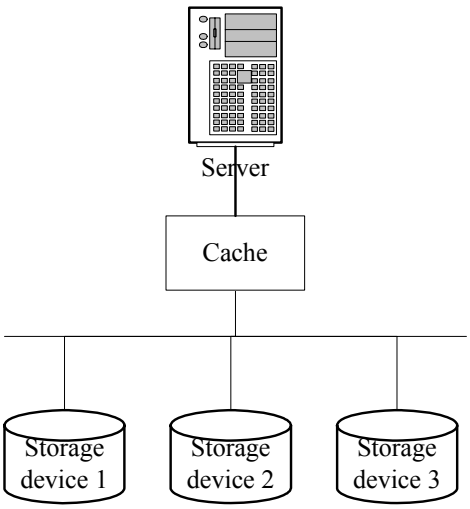

Fig 1: The Model of Hierarchical Storage System.

3.2. The Performance Formula of Storage System

The performance gain is defined to be the ratio of access times without and with cache [8].

$$g = \frac{access\ time\ without\ cache}{access\ time\ with\ cache} \qquad (1)$$

The factors influencing the I/O performance are the I/O latency of devices, the hit ratio of cache and data sizes. In order to simplify calculation, we suppose that data objects have the same sizes. On the assumption that the accessed data objects are $\{O_1, O_2, \cdots, O_n\}$, and their access frequencies are $\{m_1,$

m_2, …, m_n}, the total access frequencies of storage system is as the following:

$$M = \sum_{i=1}^{n} m_i \tag{2}$$

Without cache, accessed data comes from different storage devices, their access time are {t_1, t_2, t_3, …, t_n}, so the total access time is as following:

$$T_o = \sum_{i=1}^{n} m_i * t_i \tag{3}$$

If the storage system configures cache, suppose the cache hit ratios of objects are {p_1, p_2, …,p_n}, the total cache hit ratio is as following:

$$P = \sum_{i=1}^{n} p_i \tag{4}$$

Suppose access time of data object in cache is t_c, the total access time with cache is as following:

$$T_c = \sum_{i=1}^{n} M * p_i * t_c + \sum_{i=1}^{n} (m_i - M * p_i) * t_i \tag{5}$$

Thus performance gain is as the following:

$$g = \frac{T_o}{T_c} = \frac{\sum_{i=1}^{n} m_i * t_i}{\sum_{i=1}^{n} M * p_i * t_c + \sum_{i=1}^{n} (m_i - M * p_i) * t_i} = \frac{1}{1 - \frac{M * \sum_{i=1}^{n} p_i (t_i - t_c)}{\sum_{i=1}^{n} m_i * t_i}} \tag{6}$$

From the formula (6), the following conclusions are drawn. It is not the high total cache hit ratio but the high cache hit ratio of the data objects with long access time from devices that result to high performance gain. The usual cache replace algorithms are used to increase the total cache hit ratio, so they are no more right in the case.

If the access time from different devices is the same, the formula (6) can express as the following:

$$g = \frac{1}{1 - \frac{M * (t_i - t_c) * \sum_{i=1}^{n} p_i}{(\sum_{i=1}^{n} m_i) * t_i}} = \frac{1}{1 - \frac{(t_i - t_c) * P}{t_i}} \tag{7}$$

In this case, the higher total cache hit ratio means the higher I/O performance gain. So the usual cache replace algorithms such as LFU, LRU and OPT are based on this conclusion.

4. The Cache Replace Algorithms

There are usually two goals to design cache replace algorithms, one is to make the cache hit ratio higher to obtain least access time, and the other is to simplify the cache algorithms in order to minimize the overhead [9].

As described in section 3, the usual replace algorithms are based on the formula (7), so they are not the optimization algorithms for storage system. In this section, we propose some algorithms based on the formula (6), which are designed in order to obtain least access time but not highest cache hit ratio. In addition, we also try to simplify the cache replace algorithms in order to minimize the algorithm overhead.

4.1. The WLFU algorithm

As described above, the high cache hit ratio of the data objects with long access time from devices result to high performance gain. We define a weighting hit ratio for each object in cache as the following:

$$p_i' = \lfloor \frac{t_i}{t_{\min}} \rfloor * p_i = c_i p_i \tag{8}$$

Input: The request stream x_1, x_2, …, x_i, …
For every i≥1, the following two cases must occur.
Case 1: If i=1 then: $t_{min}=t_i$
Otherwise: If $t_i < t_{min}$ then
{$k=\lfloor t_{min} / t_i \rfloor$ and $t_{min}=t_i$ and
for every O_j in cache $p_i= p_i*k$ and $c_i= c_i*k$}
Case 2: If x_i is in cache
then: $p_i=p_i+c_i$ and sort the p queue
Otherwise:
The following two cases must occur.
1: If cache is full then:
delete object with minimal p_i
2: $c_i=\lfloor t_i / t_{min} \rfloor$ and $p_i=c_i$
and insert p in queue

Fig 2: The Weighting Least Frequently Used algorithm.

where, t_{min} is the minimal devices access time, t_i is the object's device access time, c_i is an integer to express the relative access time of O_i. We use c_i to replace t_i in order to make the weighing value simple. t_{min} is given as the following: when the first data object is accessed, its device access time is set as t_{min}, afterwards, if the device access time of data objects is not less than t_{min}, it is not changed; otherwise, the new device access time is set as t_{min}, at the same time, the weighting hit ratios of all objects in cache are multiplied by the ratio of old and new t_{min}. In the case,

the weighting cache hit ratio is used to replace the cache hit ratio, and the objects having little p_i' in cache are first replaced, which makes the object with long device access time in cache long time and improve its cache hit. So the average access time of storage system is little according to formula (6).

In order to make algorithm simple, the LFU algorithm usually replaces hit ratio with frequency, the WLFU algorithm also use the same technique. The WLFU algorithm uses a queue to record the weighting frequency of all objects in cache. If an object is not accessed in cache, when it is put in cache, its weight value is calculated and weighting frequency is set as weight value, at the same time, the t_{min} may be changed, and all of the weighting frequency and weight value maybe renewed. If an object is accessed in cache, its weighting frequency is added by weight value. So the WLFU algorithm selects the data object with the least weighting frequency as the replaced object when the cache needs storage space to cache new data object. The WLFU is described as figure 2.

This algorithm is designed based on the assumption that all data objects have the same size in storage system, but it is also suited to the storage system that has different object sizes. As described the above, all data objects have the same size in storage system, which means that the device access time is the ones of unit data object. Although the larger object can make its weighting frequency increases, it also impropriates more cache space. So in storage system with different object sizes, the device access time is replaced with the ones of unit data object. In the case the same algorithm is used. As this algorithm is used, there are many objects that have the same weighting frequency but different sizes, how to select the replaced object? Our scheme is to select the large object to be replaced because the write of large object can reduce the replace latency.

The following algorithms are also designed for storage system with same object size, but they are all suitable to storage system with different object sizes for the above reasons.

4.2. The WLRU algorithm

The WLFU algorithm captures the notion of frequency, but it pays almost no attention to recent history, and does not adapt well to varying access patterns since it accumulates stale objects with high frequency counts that may no longer be useful. The WLRU algorithm is designed to capture recency. In order to design it, we first have a look at the LRU algorithm. LRU algorithm uses recency to replace frequency, which can reduce algorithm overhead. LRU uses a recency stack to realize object queue, the top object of the LRU stack is accessed recently and the bottom object is least recently accessed. LRU selects the object at the bottom of stack to be replaced as storage system needs cache space to cache new object.

Input: The request stream $x_1, x_2, \ldots, x_i, \ldots$
Initialization: window=1
For every $i \geqslant 1$, The following four cases must occur.
Case 1: If i=1 then: $t_{min}=t_{max}= t_i$
Otherwise: one and only one of the following two cases must occur.
1: If $t_i< t_{min}$ then $t_{min}=t_i$
2: If $t_i> t_{max}$ then for every O_j in cache, $k_j= k_j*\lfloor t_i / t_{max} \rfloor$ and $t_{max}=t_i$
Case 2: $k=\lfloor t_{max} / t_{min} \rfloor$ and if k>window then window=k
Case 3: If x_i isn't in cache then: The following two cases must occur.
1: If stack is full then: delete object at the bottom of stack and replace the object in cache
2: $k_i=\lfloor t_{max} / t_i \rfloor$
Otherwise: delete x_i in the stack
Case 4: If the k_i tier is not empty then move all seriate O_j, whose local $\geqslant k_i$, down to the next tier
Put x_i in the k_i tier of the stack

Fig 3: The Weighting Least Recently Used algorithm.

Considering the device access time different, the WLRU algorithm gives a weight value k_i to the recency of each object. Precisely, if some object has long device access time, its recency is multiplied by a small weight value; otherwise, its recency is multiplied by a big weight value. k_i is integer calculated as (9).

$$r_i' = \lfloor \frac{t_{\max}}{t_i} \rfloor * r_i = k_i r_i \quad (9)$$

In order to make a queue with weighted recency, WLRU set a window at the top of the LRU stack. In the window, the new object can insert anywhere; out the window, the order of objects is no changed unless some objects are deleted. The window size is set as the integer ratio of the maximal and the minimal device access time. When an object is first put in cache, its device access time is recorded as both the maximal and the minimal device access time and the window size is set as 1. Afterwards, when some new object is put in cache, if its device access time is more(or less) than the maximal(or minimal) device access time, the maximal(or minimal)is set as the now device access time and the window size value is renewed, otherwise the window size value is not changed.

The WLRU algorithm is realized with a stack as figure 3. Case 1 is used to revise the maximal and minimal device access time. At the same time, the weight values of all objects in cache are renewed if the maximal device access time is revised. Case 2 changes the size of the window as the maximal/minimal device access time is revised. Case 3 deletes hit object or replace the weighting least recently access object in the stack. Case 4 inserts recently access object in the window according to weight value.

4.3. The WLFRU algorithm

The WLFU algorithm captures frequency and the WLRU algorithm captures recency, which utilize one part of I/O locality to improve the performance of storage system. The WLFRU algorithm is a comprehensive one that uses both frequency and recency to select the replaced object.

Input: The request stream $x_1, x_2, \ldots, x_i, \ldots$
Initialization: $L=C/2$, $window_1= window_2=1$, $k=0$
For every $i\geq1$, one and only one of following two cases must occur.
Case1: x_i is in Q_1 or Q_2: Cache hit, insert x_i according to WLRU in Q_2
Case2: x_i is neither in Q_1 nor in Q_2, Cache miss. One and only one of following two cases must occur.
1: $|Q_1|=L$, then
a) Replace the WLRU object with x_i in Q_1, and $k=k+1$
b) if $k>10$ then $L=L+1$
2: $|Q_1|<L$, then
a) If $|Q_1|+|Q_2|=C$, then delete the WLRU object in Q_2, and $k=k-1$
b) Insert x_i according to WLRU in Q_1
c) If $k<-10$ then $L=L-1$

Fig. 4: The Least Frequently/Recently Used algorithm.

In order to utilize the frequency locality, we use two variable-sized stacks Q_1 and Q_2 to record the history of the cached object. The first holds objects that have been accessed only once recently and the second holds objects that have been accessed at least twice recently, so the objects in Q_1 (or Q_2) may be least (or frequently) accessed. In order to utilize the recency locality, objects in both Q_1 and Q_2 are organized according to the WLRU. Suppose the cache size is C and the maximal size of Q_1 is L, Q_1 and Q_2 satisfy the following:

$0\leq|Q_1|\leq L$，$0\leq|Q_2|\leq C$，$0\leq|Q_1|+|Q_2|\leq C$

L is continually revised in order to improve the utility of cache, at the same time it can reduce jouncing in Q_1. For example, if some objects are continually accessed in cycle and the size of Q_1 is less than the total size of those objects, objects are frequently replaced and the cache hit is 0. In the case, L must be increased. However, if L is too big, Q_2 is small, which may result to objects frequently accessed to be replaced. The algorithm revises L according to the accumulating integer k. k is set as 0 at the beginning, and it increase (or decrease) as a replacement is taken place in Q_1 (or Q_2). L is changed only when k is accumulated to some value (such as 10), which can reduce the revising of L.

The WLFRU algorithm is described as figure 4. L is set with C/2 at beginning, which can reduce L's revising. In case 1, cache hit takes place, the hit object is inserted in Q_2 according to WLRU. In case 2, cache miss takes place, the object is inserted in Q_1 according to WLRU. Cache replace may take place in case 2. If $|Q_1|=L$, replace takes place in Q_1 regardless cache being full or not, so k increases, at the same time, L may increase as k is more than a threshold. If $|Q_1|<L$ and $|Q_1|+|Q_2|=C$, which means that L is too big, so cache replace takes place in Q_2 and k is reduced to reduce L.

5. Experiment and Results

Cache hit ratio is usually used to evaluate the performance of cache, which shows how cache can reduce device I/O. However, as described above, the higher cache hit ratio doesn't mean the less average access time in storage system. Thus we use average access time to evaluate the performance of storage system. At the same time, we also measure cache-hit ratios to compare it with average access time.

We used Disksim simulator, which was developed by Carnegie Mellon University, to simulate cache storage system. Disksim is an efficient, accurate and highly configurable disk system simulator developed to support research into various aspects of storage subsystem architecture [10]. Disksim contains a cache module that can simulate cache replace algorithms such as LRU, and we have programmed to realize algorithms such as WLFU, WLRU and WLFRU. We used the synthetic traces contained in Disksim to simulate those algorithms in order to compare their performance.

The storage system configured two disks. The average device access time was constant but the ratio of device access time was set as 1, 4 and 16. We measured both cache hit ratios and average response time of storage system with LRU, WLFU, WLRU, and WLFRU. The cache hit ratios are shown as the figure 5, WLFU and WLRU have cache hit ratio as much as LRU, and WLFRU has the highest cache hit ratio.

Since WLRU and WLFU enhance the hit ratios of some objects with more device access time while reduce the hit ratios of some objects with less device access time, it doesn't improve cache hit ratio compare to LRU. However, WLFRU uses both frequency and recency to capture locality, so its cache hit ratio is higher than the two ones.

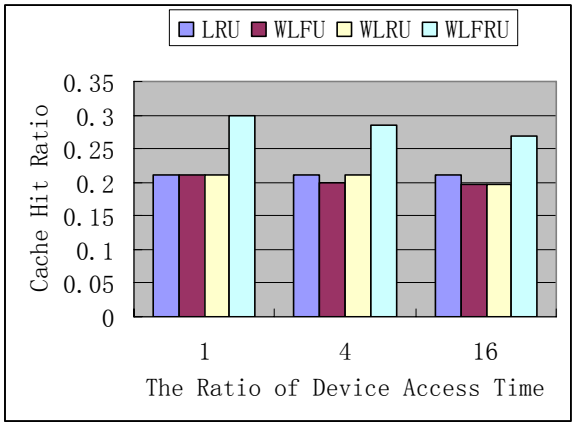

Fig. 5: The cache hit ratios vary with varying of device access time.

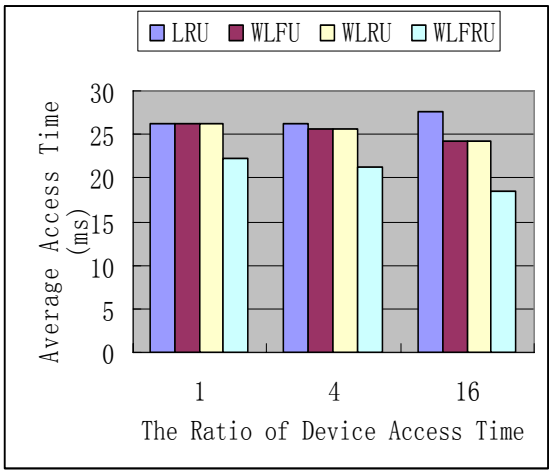

Fig. 6: The average access time vary with varying of device access time.

The average response times of storage system are described as figure 6, WLRU and WLFU have less average response time than LRU and the WLFRU has the least average response time. This is because WLFU, WLRU and WLFRU consider not only access locality but also the difference of device access time.

6. Conclusions and the Future Work

Configuring cache in storage system is an effective means to improve its performance. Considering the effect of device access time, the higher cache hit ratio doesn't mean the less average access time. Based on this conclusion, we propose WLFU, WLRU and WLFRU algorithm, which consider not only the I/O locality but also the device access time, so they have higher performance than usual algorithm. However, these algorithms don't consider the effect of network bandwidth and I/O load. Additionally, in WLFRU algorithm, although we use accumulated k to revise L, how to select the threshold with I/O load varying is not considered. In the future, we will research on cache replace algorithms adapted to the varying of network bandwidth and I/O load. We also plan to improve WLFRU algorithm to make it suit to different I/O load. Additionally, the cache algorithms about distributed cache and cooperation cache are our goal.

Acknowledgement

This work is partially supported by National Nature Science Foundation of China (Grant No. 60273073).

References

[1] W. R. Carr and J. L. Hennessy, WSClock – a simple and effective algorithm for virtual memory management. Proc. 8th Symp. Operating System Principles, pp. 87–95, 1981.

[2] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-K page replacement algorithm," *J. ACM*, 46: 92–112, 1999.

[3] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–142, 1990.

[4] Ekow Otoo, Frank Olken and Arie Shoshani. Disk Cache Replacement Algorithm for Storage Resource Managers in Data Grids. Proceedings of the IEEE/ACM SC2002 Conference, November, 2002.

[5] 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*, 50:1352–1360, 2001.

[6] Y. Zhou and J. F. Philbin, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annual Tech. Conf. (USENIX 2001), Boston, MA*, pp. 91–104, June 2001

[7] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the Second USENIX Conference on File and*

Storage Technologies (FAST), pages 115–130, San Francisco, CA, Mar. 2003.

[8] Ulrich Hahn, Werner Dilling, Dietmar Kallta. Improved Adaptive Replacement Algorithm for Disk Caches in HSM Systems. IEEE Symposium on Mass Storage Systems, March, pp. 128-140, 1999.

[9] Aameek Singh etc. A Hybrid Access Model for Storage Area Networks. Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005).

[10] John S. Bucy etc. the DiskSim Simulation Environment Version three Reference Manual. School of Computer Science, Carnegie Mellon, 2003.