ISSN 2229-5518

Proxy Caching Algorithm And The Improvements To Be Made

Jyotsna Singh
Research Scholar, MCA
Thakur Institute of Management Studies, Career Development and Research, Mumbai (MS), India

Surbhi Shukla
Research Scholar, MCA
Thakur Institute of Management Studies, Career Development and Research, Mumbai (MS), India

Abstract: **With the growing number of World Wide Web users, the constant load on servers is rapidly increasing. It puts tremendous pressure on both network load and server load. The Caching Technique has gained massive popularity because it reduces both these loads by importing copies of files from server that the client usually accesses, thereby reducing traffic. It can either be done at the client's system or in the network (by a proxy server or gateway). We assess the potential of proxy servers to cache documents retrieved with protocols like HTTP, GOPHER, FTP and WAIS World Wide Web browsers. This technique brings down the response time by fetching results comparatively faster. Proper utilization of time takes place wherein a subset of documents is selected for caching, so that a given performance metric is maximized. At the same time, the cache must ensure consistency of the cached documents. Cache consistency algorithms enforce appropriate guarantees about the staleness of the cached documents. An unified cache maintenance algorithm comes into play, namely LNC-R-WS-U, which integrates both cache replacement and consistency algorithms.**

Keywords: Proxy server, caching, caching policies, caching algorithms.

I. INTRODUCTION

In Computer Technology, A Proxy Server is a server that acts as a mediator for the requests received from the clients that demand information or resources from other servers. Also known as 'An Application Level Gateway, it acts an an intermediary between a local network and larger-scale network. Effectiveness of performance and increased security are its pros. A Caching Proxy is a function of a proxy server that caches Web Pages on the server so that the page can be quickly retrieved by the same or different user the next time that page is requested. It is also referred to as 'A Web Proxy Cache'. Proxy Caching allows a server to act as an intermediate buffer between a user and a provider of web content. When a user accesses a website, proxies reply on behalf of the originals servers, in split seconds.

Without Caching, The WWW would become a victim of its own success. An attempt to scale network and server bandwidth to keep up with client demand is an expensive strategy. An alternative to the above method is caching. Caching effectively saves copies of popular documents and migrates them from servers closer to clients. It reduces delays. Network managers see less traffic. Web servers see lesser hit rates because most of the traffic is diverted to these cache servers. A cache may be used on any of the following: a per-client basis, within networks used by the Web, or on web servers. The second solution, also known as a "proxy server" or "proxy gateway" is studied with the ability to cache documents. We use the term "caching proxy". A caching proxy's job is not an easy one. First, arrival traffic is caused as a result of the union of the URL requests of many clients. For a hit in case of a caching proxy, the same document must be requested by the user two or more times already or the same document is supposed to be requested by two users. Second, a caching proxy often operates as a second (or higher) level cache, that processes only the misses left over from Web clients that use a per-client cache or a client specific cache (e.g., Mosaic and Netscape). The misses are passed to the proxy-server from the client usually do not contain a document requested twice by the same user. The caching proxy is therefore, used for cache documents requested by two or more users. This reduces the fraction of requests that the proxy can satisfactorily retrieve from its cache, known as the *hit rate* .How do we determine the effectiveness of a caching proxy? To answer this, we must first know how much inherent duplication there is in the URLs arriving at a caching proxy. We simulate a proxy server with an infinite disk area, so that the proxy contains every document that has ever been accessed. This gives an upper bound on the hit rate that a real caching proxy can possibly ever achieve. The input to the simulation is traces of all URL accesses of three different workloads from a certain university community during a semester. Overall, we see that there is a 30%-50% hit rate. The maximum disk area required for there to be no document replacement needs to be examined. Then, we consider the case of finite disk areas, in which replacement must occur, and compare the hit rate and cache size on the basis of three replacement policies: least recently used (LRU) and two variations of LRU. LRU is shown to have quite a noticeable defect that becomes more prominent as the need and frequency

IJSER

of replacements rises. The best replacement policies are used to examine the effect on hit rate and cache size pertaining to restrictions on document sizes to cache and whether or not to cache only specific document types,sizes, or URL domains.

II. EXISTING AND PROPOSED ALGORITHMS

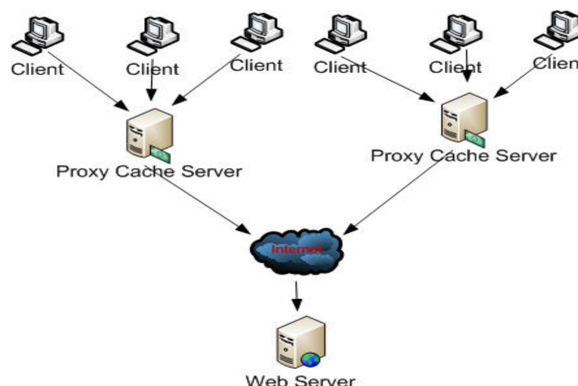
A. Existing Algorithms

Caching is done using in two forms in the Web. The first being is a *client cache*, which is built into a Web browser. A Web browser with caching stores not only the documents currently being displayed in browsers, but also documents that have been requested and accessed in the past. Client caches are of two types: persistent and non-persistent. A persistent client cache retains its documents in between invocations of the Web browser; Netscape uses a persistent cache. A non-persistent client cache (used in Mosaic) deallocates any memory space or disk usage that was being employed for caching when the user quits the browser. Client specific caches maintain consistency of cached files with server copies by issuing an optional conditional-GET to the http server or proxy-server.

The second form of caching being explored here, is in the network used by the Web (i.e., the caching proxy that was mentioned earlier). The cache is located on a machine on a path from multiple clients to multiple servers. Some examples of caching proxies include the CERN proxy server, the DEC SRC gateway, the UNIX HENSA Archive, and in local Hyper-G servers. Normally, a caching proxy is not anything like a machine that operates a WWW client or an HTTP server. It caches URLs generated by multiple clients. Hierarchical usage of caching proxies is possible, so that they cache URLs from other caching proxies. In this case, we can identify caches as first level caches, second level caches, and so on. A hierarchical arrangement is just one possible configuration.

The size and cost concerns make web caching a much more severe problem than traditional caching. Below we first summarize and take a look at a variety of web caching algorithms proposed so far.

B. Image Capture



C. Algorithms used in the existing systems

Cache algorithms (also frequently called cache replacement algorithms or cache replacement policies) are optimizing instructions—or algorithms—that a computer program or a hardware-maintained structure ought to follow to manage a cache of information stored on the computer. When the cache is full, the algorithm must make the choice as to which items to discard to make room for the new ones.

There are, in total, 17 existing caching algorithms used for caching replacements, which attempt to minimize various cost metrics, such as miss ratio, byte miss ratio, average latency, and total cost. We provide a brief overview of what these algorithms mean actually along with description of few of them. In describing the various algorithms, it is convenient to view each request for a document as being satisfied in the following way: the algorithm brings the newly requested document into the cache and then filters out the documents until the capacity of the cache is no longer exceeded. Algorithms are then discerned by how they choose among documents to evict. This view allows for the possibility that the requested document may be evicted upon its arrival into the cache itself, which means it replaces no other document in the cache.

1) OVERVIEW:

The average memory reference time is

$$T = M * Tm + Th + E \quad (1)$$ [1]

Where

T = average memory reference time

m = miss ratio = 1 - (hit ratio)

Tm = time taken to access the main memory when there is a miss (or, with multi-level cache, average memory reference time for the next-lower cache)

Th = the latency: the time to reference the cache in case of a hit

E = various secondary effects, such as queuing effects in multiprocessor systems

There are two primary figures which determine the merit of a cache: The latency, and the hit rate. There are also a number of secondary factors influencing cache performance.

The "hit ratio" of a cache is defined as the occurrence of searched - for item in the cache. More efficient replacement policies to are required keep proper track of usage information for improving the hit rate for a particular cache size.

The "latency" of a cache describes how long after requesting a desired piece of information, the cache can return that item (when there is a hit). Faster replacement algorithms/strategies typically keep track of less usage information. In the case of direct-mapped cache, there is no information—to reduce the amount of time required to update that information.

Each replacement strategy is a compromise between hit rate and latency.

Hit rate measurements are typically performed on benchmark applications. The actual hit ratio varies widely from application to application. Video and audio streaming applications often have a hit ratio close to zero, because every bit of data initially read for the very first time (a compulsory miss), utilized, and then never read or written again. Even worse, many cache algorithms (in particular, LRU) allow this streaming data to fill the cache, unnecessarily pushing out of the cache information that will be required soon (cache pollution).[2]

Other things to consider:

Items with different cost: retain items that are expensive to obtain, e.g. those that take a long time to get.

Items taking up more cache space: If items have different sizes, the cache may want to discard a large-sized chunk of data to store several smaller ones.

Items that expire with time: Some caches keep information even after it expires (e.g. web browser cache, a news cache, a DNS). The computer may get rid of items because they have expired. No further caching algorithm may be required on the basis of the size of the cache.There are also various algorithms to maintain cache coherency. This applies only to instances where *multiple* independent caches are used for the *same* data (for example many database servers updating the single shared data file)

2) *2.3.2. FEW ALGORITHMS:*

- Bélády's Algorithm

We look for an algorithm that always discards the information which will not be needed for the longest period of time in the future. Since it is generally impossible to predict how far in the future information will be needed, this is generally not advised for implementation in practice. The minimum can be calculated only after thorough analysis and experimentation, the effectiveness of the actually chosen cache algorithm can be compared.

Access Sequence	5	0	1	2	0	3	1	2	5	2
Frame1	5	5	5	2	2	2	2	2	2	2
Frame 2		0	0	0	0	3	3	3	3	3
Frame 3			1	1	1	1	1	1	5	5
	f	f	f	f		f			f	

At the moment when a page fault occurs, some set of pages is in memory. As shown in the example, the sequence of '5', '0', '1' is accessed by Frame 1, Frame 2, and Frame 3 respectively. When '2' is accessed, it replaces value '5', which is in frame 1 since it predicts that value '5' is not going to be accessed in the future. Just because a real-life general purpose operating system cannot actually figure out predict when '5' will be accessed, Bélády's Algorithm cannot be implemented on such a system.

- First in First Out (FIFO)

Using this algorithm, the cache behaves in the same way as it behaves in a FIFO queue. The cache removes the block accessed first irrespective of how many times it was accessed before.

- Last in First Out (LIFO)

With the help this algorithm, the cache behaves in the exact opposite way as a FIFO queue. Recently accessed blocks are removed irrespective of number of times they have been accessed.

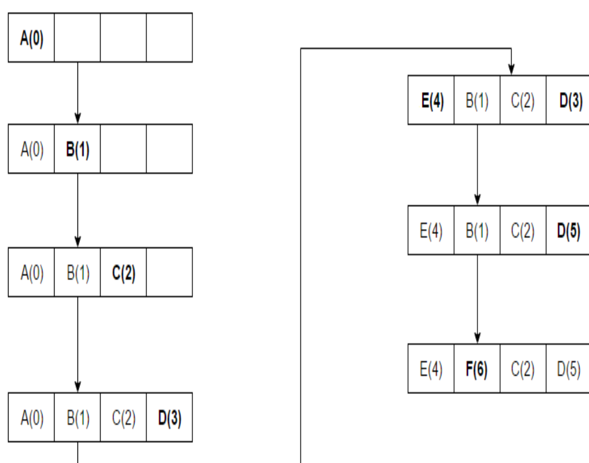
- Least Recently Used (LRU)

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is not very cost-effective if one wants to make sure the algorithm always removes the least recently used item. Implementations of this technique require "age bits" for cache-lines and then tracking the "Least Recently Used" cache-line based on age-bits. In this kind of an implementation, every time a cache-line is used, the age of all other cache-lines changes. This is a family of caching algorithms with 2Q by Betty O'Neil, Gerhard

IJSER

Weikum Theodore Johnson and Dennis Shasha ,[3], and LRU/K by Pat O'Neil, .[4].

The access sequence for the below example is A B C D E D F.

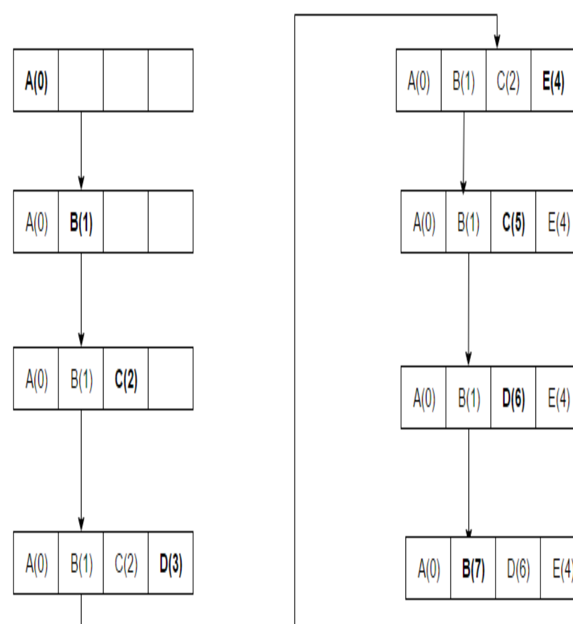


Once A B C D gets installed in the blocks with sequence numbers ,1 is incremented for each new access and when E is accessed, it is taken for a miss and needs to be installed in one of the blocks. Since A has the lowest Rank (A (0)), E will replace A.

- Most Recently Used (MRU)

In contrast to LRU, it discards the most recently used items first. From findings of the 11th VLDB conference, Chou and DeWitt noted that "When a file is being repeatedly scanned in a [Looping Sequential] reference pattern, MRU is the best replacement algorithm." For random access patterns and repeated scans over large datasets MRU cache algorithms have more hits than LRU due to their tendency to retain older data .[5], some researchers had noted. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be used.

Access sequence for the example below:A B C D E C D B.



Here, A B C D are placed in the cache as there is some still space available. At the 5th access E, the block which held D is now replaced with E as this block was used most recently. Another access to C and in the next access to D, C is replaced as it was the block accessed just before D and so on.

IJSER

- Least-Frequently Used (LFU)

It counts how often an item is needed. Items that are used least often are discarded first. This works very similar to LRU except that instead of storing the value of how recently a block was accessed, we store the value of the number of times it was accessed. So, while running an access sequence a block which was used least number of times from our cache will get replaced. e.g., if A was accessed 5 times and B was used 3 times and others C and D were used 10 times each, we will replace B.

3) *Adaptive Replacement Cache (ARC)*

Constantly balances between LRU and LFU, to bring improvements in the combined result .[6] It uses information about recently-discarded cache items to dynamically adjust the size of the protected segment and the probationary segment to make the best use of the available cache space. Adaptive replacement algorithm is explained with the example.

- Clock with Adaptive Replacement (CAR)

Combines the advantages of Adaptive Replacement Cache (ARC) and CLOCK. It has performance along the dame lines as ARC, and substantially surpasses both LRU and CLOCK in terms of performance. CAR is self-tuning and requires no user-specified magic parameters. It uses 4 doubly linked lists which includes two clocks

T1 and T2 and two simple LRU lists B1 and B2. T1 clock stores pages based on "recency" or "short term utility" and T2 stores pages with "frequency" or "long term utility". T1 and T2 contain those pages of the cache, while B1 and B2 contain pages that have recently been evicted from T1 and T2 respectively. The algorithm tries to regulate the size of these lists B1≈T2 and B2≈T1. New pages are then inserted in T1 or T2. If there is a hit in B1, size of T1 is increased and similarly if there is a hit in B2, size of T1 is decreased. The adaptation rule used has the same principle as that in ARC, rely and invest more in lists that will give more hits when more pages are added to it.

- Pannier: Container-based caching algorithm for compound objects:

Pannier[7], like mentioned is a container-based flash caching methodology which makes use of divergent (heterogeneous) containers, in which blocks held have highly varying access patterns. It uses a priority-queue based survival queue structure to rank the containers on the basis of their survival time, which is proportional to the live data in the container. Pannier is built based on Segmented LRU (S2LRU), which segregates out hot and cold data. Pannier also employs a multi-step feedback controller to throttle flash writes to ensure flash lifespan

D. *Proposed Algorithm*

After going through all the algorithms used in today's date,

We think of coming up with an algorithm that serves the following purposes:

i. Cache Maintenance: The cached data must be consistent and patched.

ii. Cache Latency: It is the amount of time that it takes for information from the cache to travel to the requested site. The cache Latency must be less compared to the algorithms mentioned above.

iii. Cache Coherency: Cache Coherency is the uniformity of shared resource data that ends up getting stored in multiple local caches. Hence, only one memory must be allocated for such data and the duplicates should be erased, removing all the redundancies.

iv. Cache Hit-rate: A cache is made up of a pool of entries. The percentage of accesses that result in cache hits or success is known as the hit rate or hit - ratio of the cache. The alternative situation, when the cache is looked up and found not to contain data with the desired tag, has become known as a cache miss.

v. Pre-Fetching: Prefetching refers to fetching information from web servers even before they are desired. The prefetching process will be highly essential for the personalization of web details.

vi. Personalization:Personalization provides the web pages as per the needs of the web users.

III. Conclusion

So as we have seen the proposed alogorithm we can achieve desired properties like i) Reliability: The algorithm should be available and reliable. Its integrity should be a high priority and should be maintained at all costs. ii)Cost-efficient: The algorithm should be cost-efficient and resource-effective. There is no point in spending effort on a system that doesn't give the expected results. iii) Usability: The algorithm must be usable and should not create barriers for the new comers to understand and implement. iv) Transparency: A Web caching system should be transparent for the user. The only results user should notice are faster response and higher availability.

REFERENCES

[1] Partitioned cache and management method for selectively caching data by type, Applicant - International Business Machines Corporation, Publication date - Jul 16, 2002, Filing date - Nov 9, 1999,

[2] "Page Placement Algorithms For Large Real-Indexed Caches", by R.E. Kessler and Mark D. Hill, University of Wisconsin. ACM Transactions on Computer Systems, vol. 10, No. 4, Nov. 1992, pp. 338-359.

[3] Theodore Johnson University of Florida Gainesville, FL 32611 Dennis Shasha Courant Institute, New York University New York, NY 10012 ' and Novell, Inc. Summit, NJ 07901, bolume 7, issue 01, March 2001, pages 350-371.

[4] Computer Networks and ISDN Systems, Volume 27, Issue 2, November 1994, Pages 165-173

[5] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. IEEE Communications Surveys & Tutorials (Volume: 6, Issue: 2, Second Quarter 2004)

[6] Computer Communications, Volume 24, Issue 2, 1 February 2001, Pages 137-143

[7] A survey of Web cache replacement strategies, ACM Computing Surveys (CSUR) Surveys Homepage archive
Volume 35 Issue 4, December 2003 Pages 374-398 ACM New York,NY, USA

IJSER