From FIFO to Predictive Cache Replacement

Daniel Meint, Stefan Liebald*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: d.meint@tum.de, liebald@net.in.tum.de

Abstract—**Caching is an important technique to accelerate data reads in various hardware and software systems. The choice of a replacement policy to decide which item to evict in order to make space for newly requested data is at the core of every cache design. A vast number of heuristics have been proposed in the literature. This paper gives an overview of some of the most popular replacement mechanisms. The strategies are classified and described. An exhaustive taxonomy of traditional strategies is proposed and explained. The paper also presents the Adaptive Replacement Cache and a predictive cache replacement strategy that was designed specifically with multimedia Web traffic characteristics in mind. Further, techniques that are closely related to the cache replacement issue, especially in Web caching, are discussed.**

Index Terms—**cache replacement strategies, network caching, adaptive replacement cache, predictive cache replacement, multimedia content delivery**

1. Introduction

Caches are employed to *localize traffic* by temporarily storing data close to the consumer and, today, they are ubiquitous in multiple areas of computing. Hardware-managed caches are used by the CPU and the GPU, for example. Software caching is found in database systems and on the Web. Generally, cached information can be retrieved significantly faster than if the origin storage would have to be consulted. However, cache capacity is typically limited and only a fraction of the existing resources can be stored at any point in time.

When data is requested, the *cache client*, such as a Web browser or a CPU, checks its closest cache first. If the information is available in the cache, we refer to this as a *cache hit*. If not, a *cache miss* occurs and data has to be read from lower-level memory. This lower-level memory can either be the original location of the resource, like a Web server, or, in case of a hierarchy of caches, a different, lower-tier cache.

When missing information needs to be brought into the cache and the cache is already full, old objects must be removed. The "victim" can be drawn randomly or chosen in a deterministic process. The respective heuristic is called the *cache replacement strategy* (also *cache replacement policy*, *eviction policy* or *removal policy* [6]) and is a central component of every caching scheme.

This paper provides an overview of extensively researched cache replacement strategies and further describes limitations and challenges of caching large objects, like multimedia content. The focus is hereby drawn towards *Web caching*.

The next section describes how cache replacement strategies can be evaluated and compared. Subsequently, some simple cache replacement approaches that partly originated from traditional disciplines of computing are discussed. Section 4 then presents the Adaptive Replacement Cache, a more recent proposal to cope with dynamic access patterns. Finally, Section 5 addresses the challenges of caching multimedia content on the Web and outlines how prediction-based replacement can be particularly effective to tackle these challenges.

2. Evaluation of Cache Replacement Policies

Mathematical models exist to evaluate the performance of cache replacement algorithms [1], [11]. Competitive Analysis compares the performance of an algorithm with the best possible performance [24]. The resulting *competitive ratio* corresponds to "the maximum ratio of the algorithms cost to the optimal offline algorithm's cost over all possible request sequences" [1]. An *offline* algorithm knows the entire request stream from the very start and can always make the best possible decision. It, therefore, only serves as a theoretical upper bound on the achievable performance by any *online* policy. Competitive Analysis merely studies worst-case scenarios and is difficult, especially when documents can be of variable size [8].

It is more common to conduct experimental studies to compare different replacement strategies. An established method is to run a *trace-driven simulation*. Performance is hereby studied on realistic cache traces, which often gives more practical insights than theoretical upper and lower bounds [8]. The following metrics are amongst the most commonly reported measures:

- **Hit Ratio (HR):** The fraction of all client-requested objects that could be served from cache, e.g. a hit rate of 10% means that one out of every ten requests resulted in a cache hit.
- **Byte Hit Ratio (BHR):** The fraction of all client-requested bytes that could be served from cache. Sometimes this value is also referred to as a "weighted" hit ratio [6]. Because it takes object size into account, BHR indicates bandwidth savings better than HR.
- **Delay Savings Ratio (DSR):** DSR reports the reduction of client-perceived latency. Its calculation

doi: 10.2313/NET-2019-06-1_06

is subject to external factors, like network congestion and server stability. Precise measurement of download delays is difficult and results often fluctuate [7].

The performance of policies in trace-driven simulations is sensitive to the character of the employed trace. Wong [11] notes that inconsistent, and even contradictory results have been reported in the literature.

3. Traditional Cache Replacement

This section describes some conventional approaches to cache replacement. First, we name factors that may influence the replacement decision. Then we present some example strategies. For the purpose of this paper, traditional strategies are divided into the following groups:

- **Key-Based Strategies** sort objects upon a *primary key*. Ties are broken using secondary, tertiary or even more keys.
- **Function-Based Strategies** incorporate multiple weighted factors, in no sequential order but concurrently in a specific function that calculates the *value* of every object. The weighting parameters may be fixed or dynamically adapting to the properties of the access stream [8].
- **Randomized Strategies** use nondeterministic algorithms to remove entries. Randomized policies typically perform worst in most scenarios [11] but also require the least resources. They are therefore primarily used in systems with severely limited processing power. Since randomized policies do not keep meaningful state information, we consider this brief introduction to be sufficient and will not discuss them any further.

Among the key-based policies, we further distinguish strategies according to which keys they actually consider. This taxonomy roughly follows and combines the proposals made by Wang in [2], Podlipnig and Boszormenyi in [7], and Balamash and Krunz in [8]. Various other classifications have been used in related work, for example in [9], [38], [54].

3.1. Influencing Factors

The following keys are universally used in cache replacement to characterize cached items and determine their utility [7]:

- **Arrival:** When was an object admitted to the cache? Typically, new items are favored to stay in the cache.
- **Recency:** When was the last request for an object? Recently accessed items are favored.
- **Frequency:** How often has an object been requested? Frequently accessed items are favored.

Network traffic has certain characteristics that suggest incorporating additional keys for replacement decisions in Web caching. CPU and disk caches are typically concerned with the management of uniformly sized blocks known as *pages*. In contrast, resources from the Web are stored as whole Web *documents* and can vary greatly in size [1], [35]. Certain items may, therefore, take up a disproportional percentage of the cache's total capacity, which should be considered by the replacement mechanism. Further, the effort of obtaining information over a network is not only correlated to the data volume, but other dependencies, such as bandwidth and distance, make the calculation of a *cost* factor more complex. Finally, cached information can also become outdated. Especially HTML resources of popular websites are updated relatively frequently [8]. The following additional keys can help Web cache replacement algorithms make more informed decisions:

- **Size:** How much space does an object occupy? Small files are favored.
- **Cost:** How expensive would it be to re-fetch an object? Possible metrics include hop count and bandwidth along the delivery path, expected latency, and monetary cost. Expensive items are favored.
- **Expiration:** When is an object presumably going to become stale? Items with a long validity are favored.

3.2. Key-based Policies

Key-based policies sort their candidates upon a primary key. If the primary key does not guarantee to determine a single clear winner, i.e., multiple objects can have identical values, a secondary key is necessary to break ties. If objects could tie on the second factor as well, a tertiary key is consulted, and so on.

First In, First Out (FIFO) is the simplest arrival-based strategy. Objects leave a queue in the order in which they arrive. Hence, this basic approach always evicts the oldest object from the cache. FIFO can easily be implemented with constant computational and optimal space overhead, and is, therefore, suitable for systems with strictly limited computational power or storage capacity. FIFO completely ignores both recency and frequency of access when making decisions. Generally, in practical applications, FIFO is significantly outperformed by policies that take these factors into account [17], [18].

The SIZE [6] strategy evicts the largest objects first with the intent to make space for multiple smaller files. Favoring small files results in a higher number of total files cached and, thus, a good file hit ratio but decreased byte hit ratio [11], [35], [53]. As mentioned earlier, BHR is closely related to bandwidth savings, so if the cache's goal is to minimize download volume from the Internet, for example, this behavior is undesirable. With a priority queue based on object size, eviction can be performed in logarithmic time.

Because the subsequent discussion of ARC in Section 4 builds on a thorough understanding of recency- and frequency-based policies, we discuss these concepts in the following dedicated subsections.

3.2.1. Recency-based Policies. Recency-based policies sort objects according to how recently they were requested. The underlying rationale is that recently accessed information is more likely to be demanded again in the near future and should, therefore, be retained in the cache.

Information that has not been useful for the longest time is accordingly regarded as least valuable and should be evicted first.

For this rationale to hold, access streams need to exhibit *temporal locality*, i.e., the past and the future must not be independent, but resources become "hot" (accessed frequently) and cool down again [7], [20]. Jin and Bestavros find that temporal correlations are present in Web traffic and exist most dominantly in the *short-term* [37], [38]. Recency-based strategies are therefore most effective in caches of small sizes and high activity [11].

Caches may form a multi-level hierarchy. Rajan and Ramaswamy describe how temporal locality is inherent to first-level caches but decreases throughout a storage hierarchy in [26]. The most recently accessed files are assumed to be served from the first-level cache anyway and so, for lower-tier caches further down the delivery path, e.g. in root-level proxies, locality features are already "filtered out". Here, other heuristics can perform better than recency. Busari and Williamson demonstrate that size-sensitive policies are more effective in root-level caches in [53].

The *Least Recently Used* (LRU) strategy can be seen as the originator of recency-based replacement. It simply evicts the object that was not accessed for the longest time. Assuming requests are processed sequentially, every request has a unique timestamp. Hence, we do not need additional tie-breaker keys. LRU can be implemented with a linked-list data structure supported by a hashing mechanism for lookup. Its simplicity and constant time complexity make LRU particularly attractive for hardware caches.

LRU's biggest threats are large sequential reads of data that is only needed once and then never accessed again. Since LRU only takes recency into account, it degenerates to FIFO and these "scan" sequences quickly flush out potentially more valuable items and pollute the cache.

Many variants of LRU have been proposed. One strategy proposed by Pitkow and Recker [25] uses a different time granularity. The authors observe that client interests change on a daily basis. As a consequence, they suggest using the number of full days since the last request as a primary factor. If there are no objects older than one day, size serves as a secondary key. The largest files are replaced first.

LRU-Threshold [27] rejects files larger than a specified threshold before they can even get into the cache. It could, therefore, be argued that the primary key is size. When cached files need to be removed, the victim is determined according to LRU. Hence, we consider the actual replacement process to be recency-based. This classification agrees with the suggestions made in [11].

Other recency-based strategies include LRU* [28], LRU-Hot [29], Segmented LRU (SLRU) [30] and HLRU [32].

3.2.2. Frequency-based Policies. Frequency-based policies sort objects according to how often they have been requested in the past. The underlying rationale is that some data is consistently more popular than other data, i.e., exhibits *long-term* popularity, and information that was frequently requested in the past will keep being accessed a lot in the future. Tiebreaker policies are unavoidable because multiple objects can easily have identical frequency values.

Among Web documents, popularity distribution follows Zipf's law [4], [33]. This means that only a small set of very popular items accounts for a significant fraction of the overall traffic. Keeping the "hottest" items in the cache should be sufficient to satisfy most requests. However, when the set of popular data changes abruptly, frequency-based strategies cannot adapt as quickly as recency-based strategies [11]. Frequency-based heuristics perform best in environments with static popularity characteristics, i.e., popular data stays popular and unpopular data stays unpopular [7].

Frequency-based strategies are normally implemented with a priority queue [7] offering logarithmic time complexity per operation.

The *Least Frequently Used* (LFU) strategy always evicts the item with the lowest frequency count. Podlipnig and Boszormenyi distinguish between two forms in [7].

- **Perfect LFU** keeps track of all requests to objects ever recorded. This form is of theoretical value, but unbound space overhead makes its implementation infeasible.
- **In-Cache LFU** keeps track of requests to currently cached objects only. Once an item is removed, the count is forgotten. This form has imperfect information, but the space overhead stays manageable.

Unlike LRU, LFU is *scan-resistant*. Frequency is of relevance, so sequentially accessed items only replace each other.

However, especially In-Cache LFU suffers from a different cache pollution problem. Objects that were popular at one point in the past, and have accumulated high reference counts, hardly get flushed out, even if they are currently unpopular. Newer, potentially more useful (hotter) items have a hard time to stay in the cache because they start off with the lowest possible score. Again, this suggests that LFU performs best in systems where the demand for an object stays at a constant level.

More versatile alternatives avoid the cache pollution problem by also incorporating recency into the decision-making process. So-called *aging* mechanisms effectively decrease an object's value when it has not been requested for a certain amount of time. Recently accessed items are consequently regarded as more valuable and remain in the cache. *LFU with Dynamic Aging* (LFU-DA) [31] is an example of a hybrid strategy that combines frequency and recency aspects.

3.3. Function-based Policies

Function-based removal policies consider multiple attributes at once and not separately. So there are no more primary and secondary keys but instead a single, usually nonnegative value H associated with each cached object i.

One popular member of this category is the *GreedyDual-Size* (GD-Size or GDS) algorithm [1]. When an item enters the cache, its value is initialized as follows.

$$H_i = cost_i / size_i$$

When the cache has filled up, the item with the lowest value $H_{min} = min_{j \in cache} H_j$ is removed. Subsequently, the values of all remaining items get reduced by H_{min}:

$$\forall i : H_i \leftarrow H_i - H_{min}$$

So their scores shrink over time. On a cache hit, the value of the re-requested item gets reset to its original one.

The $cost$ parameter can be defined in various ways, depending on the objective of the caching system [8], [34]. For example, setting $cost_i = 1$ uniformly for all objects means an item's initial H value corresponds to $1/size_i$. Due to this direct inverse correlation of value and size, the largest files are regarded as least sustainable and get removed in favor of multiple smaller files. Consequently, this definition of cost maximizes the total number of files cached and, therefore, results in the best hit rate [35].

Depending on the environment, other cost definitions are possible. For network caching, GD-Size might instead define cost as the number of packets involved in a retrieval, the expected latency increase or the number of hops between the cache and the origin server [10].

The GreedyDual-Size policy is an extension of the original GreedyDual algorithm introduced by Young [36]. Today, an entire range of algorithms, referred to as the "GreedyDual-family" [10] exists. Members include GD-Frequency, GD-Size-Frequency [31], and GD*, an adaptive generalization of GD-Size [38].

Some function-based policies allow parameters to adapt to workload characteristics [7]. This means that when requests are observed to have high temporal correlations, for example, recency should be weight heavier than other factors. If the access stream indicates that popularity levels are rather steady, on the other hand, the underlying function should put more emphasis on frequency counts. The downside of this adaptive functionality is increased computational complexity [7], which could hurt performance overall, especially when adaptiveness is unnecessary.

4. Adaptive Cache Replacement

This section presents the *Adaptive Replacement Cache* (ARC) [12]. ARC neither associates H values with entries like function-based approaches nor should it be categorized as key-based in the traditional sense as presented above, since it does not apply a fixed sequence of primary key, secondary key, etc. Instead, it constantly reacts to changes in the character of the processed requests to balance recency and frequency aspects in a *self-tuning* manner, i.e., there are no parameters that need to be set manually.

ARC cleverly combines two LRU-lists of varying size and a history of recently evicted items to make removal decisions.

When an item enters the cache, it is placed at the *most recently used* (MRU) position of the recency-ordered list L_1. L_1 is therefore considered to contain recently required items. If the item gets requested a second time, while cached, it is considered to be frequently accessed, and "promoted" to a second list L_2, again, entering at the MRU position.

Both lists are further split into a "top" and "bottom" part, that is, L_1 consists of two sublists T_1 and B_1 and L_2 is the union of T_2 and B_2.

The top sections form the *main cache* and store full objects that can be returned as expected.

The bottom sections form the *ghost cache* and only store identifiers, i.e., metadata, for the objects that once were in the main cache, but got flushed out. The ghost cache merely serves a history function. It cannot provide the resource data to answer client requests.

Now, the lists T_1 containing items that have been accessed once recently and T_2 containing items that were requested at least twice compete for cache capacity. Depending on the workload attributes they grow and shrink constantly. As the full-fledged balancing process is fairly complex, we will only explain the basic idea underlying ARC's adaptability and refer to [12] for a detailed description. Intuitively, when a request reaches the cache, the following events influence ARC's behavior.

If the requested item is not contained in the main cache, but a "ghost hit" in B_1 occurs, ARC concludes that recency features are currently important. The request cannot be satisfied from cache directly because the item was pushed out of T_1 into the ghost section, only retaining metadata. If T_1 was granted more capacity, the request might have resulted in a "real" cache hit. Therefore, T_1 grows and T_2 shrinks, i.e., the least recently used item in L_2 will be evicted next.

If the request hits the B_2 ghost cache, ARC considers frequency aspects to currently be neglected. Ghost hits in B_2 therefore let T_2 grow at the expense of T_1, i.e., the least recently used item in L_1 will be evicted next.

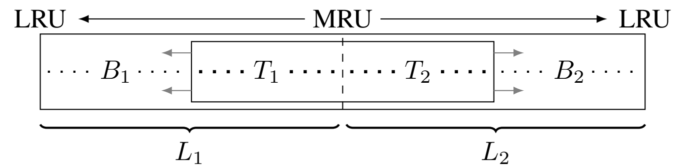

Figure 1. Simplified ARC cache directory visualized in a balanced state. The inner rectangle represents the actual cache, which is fixed in size but can freely move across the history sections. Items enter the cache at the center and get gradually pushed outwards unless they are re-accessed.

ARC is considered scan-resistant since the separation of items that have been accessed only once ($L1$) and items that have been accessed at least twice recently ($L2$) protects the latter section of being flooded by single-access streams.

It was reported to reliably and substantially outperform established mechanisms like LRU in trace-driven simulations [12], [13]. ARC does not require significantly more space than LRU and has the same, constant time complexity.

Since 2006, IBM holds a patent for the Adaptive Replacement Cache [15], which has complicated its deployment for third parties [16].

5. Predictive Cache Replacement for Web-Based Video Content

The above-described algorithms were mostly designed as general-purpose policies. LRU and GDS have been

titled “good enough” or “champion” algorithms [1], [34] because they perform sufficiently well on the basic performance metrics for caches of various sizes and environments. Wong states that cache replacement in its general form is a “solved topic” [11] and further research would only lead to marginal improvements.

However, more recently, authors have also depicted that “the network environment is dynamic and uncertain” [34] and future requirements may very well overwhelm the identified “good enough” algorithms. Podlipnig and Boszormenyi say, their sufficiency for managing multimedia content is “questionable” [7].

This section presents a more specialized caching approach to cope with the challenges of serving multimedia data on the World Wide Web. Video streaming is arguably the most extreme cases. It combines audio and imagery, requiring storage volumes and retrieval bandwidths multiple magnitudes greater than ordinary text and small graphics. The focus of this section is, therefore, on Video-on-Demand (VoD) services. First, the unique properties of video data and its consumption are described. The second subsection then presents a novel cache replacement proposal by Famaey et al. [51] that predicts the future popularity of multimedia content.

5.1. Multimedia Caching

From a storage perspective, multimedia files differ from text-based data most notably in volume. Cached objects are significantly larger in size, as compared to when cached information is text-based [11], [39]. Thus, we expect space for storage and bandwidth for transport to be the critical factors for multimedia applications. Further, caching mechanisms need to guarantee continuous delivery of video without stutters for enjoyable consumption. Dan and Sitaram conclude that traditional replacement is insufficient for video caching requirements [39].

On the other hand, multimedia data offers the possibility of significantly reducing volume “without sacrificing too much quality” [7]. Text files typically need to be compressed lossless to allow perfect reconstruction. In video compression, inter-frame techniques eliminate redundancy without compromising quality [41]. Even noticeable quality losses may sometimes be acceptable when this results in a smoother presentation.

Converting files from one representation to another, e.g. format conversion or compression, is called transcoding [42]. On the Web, *transcoding proxy servers* [42], [45] take these options into account. A proxy on the delivery path somewhere between the client and the origin Web server temporarily stores requested objects locally and acts as a cache for future requests. The so-called *Soft Caching* [44] approach allows for modification of these resources. Specifically, it allows the proxy to recode images to lower resolution versions and discard original files to save space. Upon a request, the proxy cache might then initially serve the transcoded object with lower image resolution until the original version becomes available.

This possibility adds another level of complexity to the replacement process because a replacement policy no longer only makes a binary decision on whether to completely evict an object or not, but now also needs to evaluate, if storing a copy with reduced quality instead of the original file is beneficial to the overall user experience [7], which is refered to as a *soft decision* [44].

So far, we assumed data to only ever be cached-in when it is demanded and not already in the cache. Under this *demand fetching* model, the replacement policy is the only algorithm of interest [14]. Studies have reported that, due to many single-access requests, the cache hit rate is bound to approximately 50%, even if replacement decisions would always be made optimally by some hypothetical omniscient policy [2], [27].

To lift this bound, document requests must be anticipated and files loaded into the cache before they are eventually demanded. We refer to this as *anticipatory fetching* or *prefetching*. The prediction of future requests should be done accurately and prudently because fetching unnecessary data that will never be needed can significantly increase network traffic and thereby introduce delays [45]. Trace-driven simulations have shown that prefetching data from Web servers into client caches can reduce user-perceived latency by up to 45%, but also doubles the total load on the network [46]. Prefetching is not a cache replacement issue, but the two mechanisms are closely related and cooperatively manage the cache content. When streaming video content over the Internet, preloading sequences that are about to be shown is essential to prevent disruptions in playback. The term “buffering” is often used synonymously to prefetching in the context of VoD applications [39], [40], [47].

Long videos, e.g. movies, are typically consumed linearly from beginning to end, which makes predicting the next requested frames relatively straightforward. The most critical data are, therefore, the early frames of videos. The beginnings of videos are also overall the most popular parts [48], [51]. Sen et al. [47] propose that proxy caches should store a prefix of every audio or video stream, instead of storing the entire object.

Multimedia content popularity is highly dynamic [51], but researchers have identified access patterns that make its prediction feasible. VoD customers are more likely to consume full-length movies in the evening and on the weekend, for example, creating exploitable patterns repeating on a daily and weekly basis [51]. Other research suggests that on video-sharing platforms such as YouTube, popularity patterns differ among categories. Copyright protected material, e.g. a music video, gets a significantly higher percentage of the total views on its first days online, as compared to uncopyrighted videos, e.g. user-generated video blogs, that show steadier request rates on average [49].

5.2. Predictive Least Frequently Used

The traditional LFU policy was presented in Section 3.2.2 and evicts the item that was accessed least often in the past. This section describes *Predictive Least Frequently Used* (P-LFU). P-LFU evicts the item with the lowest predicted number of requests within a specifiable prediction window. For P-LFU to perform well, accurate prediction values are vital. Prediction happens in a separate phase, prior to the actual eviction phase. Famaey, Iterbeke, Wauters and De Turck propose a generic popularity prediction algorithm and find that Web objects can be grouped according to how their popularity evolves

over time. Only four distributions are needed to cover the majority of request patterns for these items [51].

- **Constant** models steady access rates, e.g. for permanently unpopular items.
- **Power-Law** models abrupt steep changes in popularity often seen in multimedia systems.
- **Exponential** models slower changes than power-law and has shown to give the most accurate predictions out of the four distributions when applicable [51], [52].
- **Gaussian** models S-shaped request patterns. Starting off constant, a sudden significant change in popularity appears and then returns to a constant access rate.

Finding the best parameters to fit these four models to the history is a non-linear optimization problem. The authors use the Levenberg-Marquardt algorithm [50] for this purpose. Finally, the distribution with the "best fit", e.g. the one with the smallest *mean squared error* (MSE), is selected and used to make a projection on future request rates.

Famaey et al. conducted simulations on the request traces of the "VoD service of a leading European telecom operator" [51]. 75013 requests were recorded for 4971 different movies. The results indicate that P-LFU can realistically perform approximately 10% better than traditional LFU in terms of hit rate. For accurate predictions, the number of historical data points should not be smaller than 10, however [51].

Further, the algorithm predicted accesses to unpopular movies with significantly higher accuracy than requests for popular items [51].

6. Conclusion and Future Work

There exist plenty of cache replacement proposals beyond what is covered in this paper. Section 3 looked at traditional cache replacement approaches and classified them based on some commonly used factors of cacheable items, like recency and frequency of access that influence the removal process. The Adaptive Replacement Cache (ARC) was presented in Section 4 to demonstrate that improvement over LRU is possible, even without introducing unreasonable overhead. Section 5 described how multimedia Web traffic could pose difficulties for so-called "good enough" replacement strategies in the future. Some unique characteristics of multimedia content were explained and, finally, a prediction-based variant of the LFU scheme that was designed to cope with the challenges of multimedia caching was depicted.

Section 5.1 already touched upon the possibility of prefetching data into the cache before it is actually needed. Prefetching and similar mechanisms should be studied further in the context of Web caching to investigate latency reductions beyond what pure replacement strategies can achieve. Further, transcoding techniques for data reduction might render especially helpful for environments where caches are of smaller size, e.g. mobile systems. How to optimally handle the trade-off between quality and speed that transcoding caches have to deal with is another open issue to be researched.

References

[1] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," Proc. 1997 USENIX Symp. Internet Tech. and Sys., 1997, pp. 193–206.

[2] J. Wang, "A Survey of Web Caching Schemes for the Internet," ACM Comp. Commun. Review, vol. 29, no. 5, Oct. 1999, pp. 36–46.

[3] A. Tanenbaum, "Modern Operating Systems, Third Edition," Prentice Hall, Inc, 2009.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," Proc. IEEE INFOCOM Conf., 1999, pp. 126–34.

[5] H. Bahn, K. Koh, S.H. Noh, and S.M. Lyul, "Efficient Replacement of Nonuniform Objects in Web Caches," IEEE Comp., June 2002, pp. 65–73.

[6] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, and E.A. Fox, "Removal Policies in Network Caches for World Wide Web Documents," Proc. ACM SIGCOMM Conf., Stanford University, Aug. 1996, pp. 293–305.

[7] S. Podlipnig and L. Boszormenyi, "A Survey of Web Cache Replacement Strategies," ACM Comp. Surveys, vol. 35, no. 4, Dec. 2003, pp. 374–98.

[8] A. Balamash and M. Krunz, "An Overview of Web Caching Replacement Algorithms," IEEE Commun. Surveys & Tutorials, vol. 6, no. 2, 2004.

[9] C. Aggarwal, J. Wolf, and P. Fellow, "Caching on the World Wide Web," IEEE Trans. Knowledge and Data Eng., vol. 11, no. 1, Jan./Feb. 1999, pp. 94–107.

[10] G. Kastaniotis, E. Maragos, V. Dimitsas, C. Douligeris, and D.K. Despotis, "Web Proxy Caching Object Replacement: Frontier Analysis to Discover the 'Good-Enough' Algorithms," Proc. IEEE 15th International Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007, pp. 132–137.

[11] A.K.Y. Wong, "Web Cache Replacement Policies: A Pragmatic Approach," IEEE Network magazine, vol. 20, no. 1, 2006, pp. 28–34.

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

[13] N. Megiddo and D. Modha, "Outperforming LRU with an adaptive replacement cache algorithm" Computer, vol. 37, no. 4, 2004, pp. 58–65.

[14] N. Megiddo and D. Modha, "One up on LRU," login – The Magazine of the USENIX Association, vol. 28, 2003, pp. 7–11.

[15] N. Megiddo and D. Modha, "System and Method for Implementing an Adaptive Replacement Cache Policy," US Patent 6,996,676, Feb. 2006.

[16] E. Mustain, "The Saga of the ARC Algorithm and Patent," Postgresql General Bits, 2005, http://www.varlena.com/GeneralBits/96.php (accessed September 27, 2018).

[17] M. Chrobak and J. Noga, "LRU is Better than FIFO," Algorithmica, vol. 23, no. 2, 199, pp. 18—185.

[18] G. Rexha, E. Elmazi, and I. Tafa, "A Comparison of Three Page Replacement Algorithms: FIFO, LRU and Optimal," Academic Journal of Interdisciplinary Studies, vol. 4, no. 2, 2015, p. 56.

[19] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," Proc. ACM Sigmetrics Conf., ACM Press, 2002.

[20] T. Johnson and D. Shasha, "2Q: A Low Overhead High-Performance Buffer Management Replacement Algorithm," Proc. VLDB Conf., Morgan Kaufmann, 1994, pp. 297–306.

[21] 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, 2001, pp. 1352–1360.

[22] T. Saemundsson, "An Experimental Comparison of Cache Algorithms," 2012.

[23] B.D. Davison, "A Web Caching Primer," IEEE Internet Comput. 5, no. 4, 2001, pp. 38–45.

[24] D. Sleator and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," Communications of the ACM, vol. 28, no. 2, 1985, pp. 202–208.

[25] J.E. Pitkow and M. Recker, "A Simple yet Robust Caching Algorithm Based on Dynamic Access Patterns," Proc. 2nd International World Wide Web Conf., 1994, pp. 1039–1046.

[26] K. Rajan and G. Ramaswamy, "Emulating Optimal Replacement with a Shepherd Cache," Proc. 40th International Symp. on Microarchitecture, 2007.

[27] M. Abrams, C.R. Standbridge, G. Abdulla, S. Williams, and E.A. Fox, "Caching Proxies: Limitations and Potentials," Proc. 4th International International World Wide Web Conf., 1995.

[28] C.-Y. Chang, T. McGregor, and G. Holmes, "The LRU* WWW Proxy Cache Document Replacement Algorithm," Proc. Asia Pacific Web Conf., 1999.

[29] J.-M. Menaud, V. Issarny, and M. Banatre, "Improving Effectiveness of Web Caching," Recent Advances in Distributed Systems, 2000.

[30] M. Arlitt, R. Friedrich, and T. Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies," Tech. Rep. HPL-98-97(R.1), Hewlett-Packard Company, 1999.

[31] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," ACM SIGMETRICS Performance Evaluation Reviews, vol. 27, no. 4, pp. 3–11, March 2000.

[32] A.I. Vakali, "LRU-based Algorithms for Web Cache Replacement," International Conf. on Electronic Commerce and Web Technologies, 2000.

[33] E. Friedlander and V. Aggarwal, "Generalization of LRU Cache Replacement Policy with Applications to Video Streaming," 2018.

[34] J. Du, S. Gao, J. Lv, Q. Li, and S. Ma, "A Web Cache Replacement Strategy for Safety-Critical Systems," Tehnicki Vjesnik, vol. 25, no. 3, 2018, pp. 820–830.

[35] L. Cherkasova and G. Ciardo, "Role of Aging, Frequency, and Size in Web Cache Replacement Policies," International Conf. on High-Performance Computing and Networking, 2001, pp. 114–123.

[36] N. Young, "On-line caching as cache size varies," 2nd Annual ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 241–250.

[37] S. Jin and A. Bestavros, "Sources and Characteristics of Web Temporal Locality," Proc. IEEE 8th International Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2000, pp. 28–35.

[38] S. Jin and A. Bestavros, "GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams," Proc. 5th International Web Caching and Content Delivery Workshop, 2000.

[39] A. Dan and D. Sitaram, "Multimedia Caching Strategies for Heterogeneous Application and Server Environments," Multimedia Tools and Applications, vol. 4, no. 3, 1997, pp. 279–312.

[40] A. Dan, D. Dias, R. Mukherjee, D. Sitaram, and R. Tewari, "Buffering and caching in large scale video servers," Proc. IEEE CompCon, 1995, pp. 217–224.

[41] Wikipedia contributors, "Data compression," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Data_compression&oldid=860709233 (accessed September 27, 2018).

[42] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas, "Dynamic Adaptation in an Image Transcoding Proxy For Mobile Web Browsing," IEEE Personal Communications, vol. 5, no. 6, 1998, pp. 8–17.

[43] K.H. Yeung, C.C. Wong, and K.Y. Wong, "A Cache Replacement Policy for Transcoding Proxy," IEICE Trans. Commun., vol. E87-B, no. 1, 2004, pp. 209–11.

[44] J. Kangasharju, Y.G. Kwon, and A. Ortega, "Design and Implementation of a Soft Caching Proxy," Computer Networks and ISDN Systems, vol. 30, no. 22-23, 1998, pp. 2113–2121.

[45] M. Crovella and P. Batford, "The Network Effects of Prefetching," Proc. Infocom'98.

[46] V.N. Padmanabhan and J.C. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," ACM SIGCOMM Computer Communication Review, vol. 26, no. 3, 1996, pp. 22–36.

[47] S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams," Proc. Infocom'99.

[48] H. Guo, G. Shen, Z. Wang, and S. Li, "Optimized Streaming Media Proxy and its Applications", Journal of Network and Computer Applications, vol. 30, no. 1, 2007, pp. 265–281.

[49] F. Figueiredo, F. Benevenuto, and J.M. Almeida, "The Tube Over Time: Characterizing Popularity Growth of YouTube Videos," Fourth ACM International Conf. on Web Search and Data Mining, 2011, pp. 745–754.

[50] K. Levenberg, "A Method for the Solution of Certain Non-Linear Problems in Least Squares," Quarterly Journal of Applied Mathmatics, vol. 2, no. 2, 1944, pp. 164–168.

[51] J. Famaey, F. Iterbeke, T. Wauters, F. DeTurck, "Towards a Predictive Cache Replacement Strategy for Multimedia Content," Journal of Network and Computer Applications, vol. 36, no. 1, 2013, pp. 21–227.

[52] A. Tatar, M.D. de Amorim, S. Fdida, and P. Antoniadis, "A Survey on Predicting the Popularity of Web Content," Springer J. Internet Services and Applications, vol. 5, no. 1, 2014, pp. 1–20.

[53] M. Busari and C. Williamson, "On the Sensitivity of Web Proxy Cache Performance to Workload Characteristics," Proc. IEEE INFOCOM, vol. 3, 2001, pp. 1225–34.

[54] B Krishnamurthy and J. Rexford, "Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement, " Addison-Wesley Professional, 2001.