An Enhanced Semantic-based Cache Replacement Algorithm for Web Systems

1st Xuan Tung Hoang
Faculty of Information Technology
VNU University of Engineering and Technology
Hanoi, Vietnam
tunghx@vnu.edu.vn

2nd Ngoc Dung Bui
Faculty of Information Technology
University of Transport and Communications
Hanoi, Vietnam
dnbui@utc.edu.vn

***Abstract*—As Web traffics is increasing on the Internet, caching solutions for Web systems are becoming more important since they can greatly expand system scalability. An important part of a caching solution is cache replacement policy, which is responsible for selecting victim items that should be removed in order to make space for new objects. Typical replacement policies used in practice only take advantage of temporal reference locality by removing the least recently/frequently requested items from the cache. Although those policies work well in memory or filesystem cache, they are inefficient for Web systems since they do not exploit semantic relationship between Web items. This paper presents a semantic-aware caching policy that can be used in Web systems to enhance scalability. The proposed caching mechanism defines semantic distance from a web page to a set of pivot pages and use the semantic distances as a metric for choosing victims. Also, it use a function-based metric that combines access frequency and cache item size for tie-breaking. Our simulations show that out enhancements outperform traditional methods in terms of hit rate, which can be useful for websites with many small and similar-in-size web objects.**

***Index Terms*—web cache, replacement algorithm, semantic-aware, semantic distance, web performance**

I. Introduction

In recent years, web-based systems have become an essential tool for interaction among people and for providing a wide range of Internet-based services, including shopping, banking, entertainment, etc. As a consequence, the volume of transported Internet traffic has been increasing rapidly. Such growth has made the network prone to congestion and has increased the load on servers, resulting in an increase in the access times of web documents. Web caching provides an efficient solution to reduce server load by bringing documents closer to clients. As summarized in [1], caching function can be deployed at various points in the Internet: within the client browser, at or near the server (reverse proxy) to reduce the server load, or at a proxy server. A proxy server is a computer that is often placed near a gateway to the Internet and that provides a shared cache to a set of clients. Client requests arrive at the proxy regardless of the Web servers that host the required documents. The proxy either serves these requests using previously cached responses or obtains the required documents from the original Web servers on behalf of the clients. It optionally stores the responses in its cache for future use. Hence, the goals of proxy caching are twofold: first, proxy caching reduces the access latency for a document; second, it reduces the amount of external traffic that is transported over the wide-area network (primarily from servers to clients), which also reduces the users perceived latency. A proxy cache may have limited storage in which it stores popular documents that users tend to request more frequently than other documents.

978-1-5386-9313-1/19/$31.00 ©2019 IEEE

Caching policies for traditional memory systems do not necessarily perform well when applied to Web proxy cache servers for the following reasons:

- In memory systems, caches deal mostly with fixed-size pages, so the size of the page does not play any role in the replacement policy. In contrast, web documents are of variable size, and document size can affect the performance of the policy.
- The cost of retrieving missed web documents from their original servers depends on several factors, including the distance between the proxy and the original servers, the size of the document, and the bandwidth between the proxy and the original servers. Such dependence does not exist in traditional memory systems.
- Web documents are frequently updated, which means that it is very important to consider the document expiration date at replacement instances. In memory systems, pages are not generally associated with expiration dates.
- The popularity of web documents generally follows a Zipf-like law (i.e., the relative access frequency for a document is inversely proportional to the rank of that document). This essentially says that popular WWW documents are very popular and a few popular documents account for a high percentage of the overall traffic. Accordingly, document popularity needs to be considered in any Web caching policy to optimize a desired performance metric. A Zipf-like law has not been noticed in memory systems.

The design of an efficient cache replacement algorithm is crucial for caching mechanisms [2]. Since cache's storage has limited size, an intelligent mechanism is required to manage the Web cache content efficiently. It is expected that the cache content only keeps frequently accessed Web items, and

rarely accessed items should be evicted from the cache as soon as possible. The traditional caching replacement policies (e.g. [3]–[5]) are not efficient in the Web caching since they consider a single factor (such as least-recently-used factor) and ignore other factors that have impact on the efficiency of the Web caching. In these caching policies, cache pollution may happen, in which a large portion of cache content are occupied by objects that are requested rarely. Combination of various factors into one using a formula that balances their importance, to a certain extend, can reduce cache pollution and improve performance of traditional caching replacement policies. However, as point out in [5], the importance of a factor varies in different environments and applications. This creates difficulties in finding optimal combinations of cache factors for all situations. Also, since Web objects are semantically related (e.g.: Web pages are referenced among each other via HTML links), cache performance can improve greatly if semantic information is efficiently used.

This research contributes to the studies of cache replacement algorithm, especially for web caching. We propose an approach that uses (i) semantic-related techniques and (ii) function-based cache values that are calculated from access frequencies and cached items' sizes. Our approach is designed to reduce cache pollution, and thus, can improve cache hit rate in comparison with existing caching approaches. By improving higher cache hit rates, our proposed caching scheme is suitable for online news or online music websites whose contents contain many small and relatively equal web items.

The rest of this paper is structured as follows. In section II, we provide some related works on caching policy and web caching, especially semantic-aware algorithms. In section III, we proposed our algorithm. We present the performance evaluation in section 4. Finally, we conclude this paper in section 5.

II. Related Work

Due to the importance of cache replacement algorithms for web caching systems, a huge amount of work in the area can be found in literature. According to [6], these algorithms can be grouped in two categories: key-based algorithms, function-based algorithm. Also, recent research attempts show that semantic information of Web items can be used for cache replacement policies. Thus, cache replacement schemes can also be classified into semantic-based and semantic-less algorithms.

A. Key-based Algorithms

Most popular group of replacement policies are key-based. In these policies, keys are used in the decision-making for choosing victims in a prioritized fashion. A key is a simple parameter associated with each cache entry, such as age, size, or access frequency. A primary key is used to decide which item to evict from the cache in case of cache saturation. Additional keys may be used for tie-breaking in case ties happen during the selection process. Table I shows some regularly used keys in caching policy.

TABLE I
Commonly used parameters (keys) in cache replacement policies

Factor	Parameter	Rationale
Recency	*Last access time*	*Web traffic exhibits strong temporal locality.*
Frequency	*Number of previous accesses*	*Frequently accessed documents are likely to be accessed in the near future*
Cost	*Average fetching (download) delay*	*Caching documents with high fetching (download) delay can reduce the average access latency.*
Size	*Object size*	*Caching small documents can increase the hit ratio.*

Classical replacement policies, such as the LRU and the least frequently used (LFU) policies, fall under this category. LRU evicts the least recently accessed document first, on the basis that the traffic exhibits temporal locality. In other words, the further in time a document has last been requested, the less likely it will be requested in the near future. LFU evicts the least frequently accessed document first, on the basis that a popular document tends to have a long-term popularity profile. Other key-based policies (e.g., SIZE [6] and LOG2-SIZE [7]) consider document size as the primary key (large documents are evicted first), assuming that users are less likely to re-access large documents because of the high access delay associated with such documents. SIZE considers the document size as the only key, while LOG2-SIZE breaks ties according to $\lfloor log_2(DocumentSize) \rfloor$, using the last access time as a secondary key. Note that LOG2-SIZE is less sensitive than SIZE to small variations in document size (e.g. $\lfloor log_2 1024 \rfloor = \lfloor log_2 2040 \rfloor = 10$). The LRU- threshold and the LRU-MIN [7] policies are variations of the LRU policy. LRU-threshold works the same way as LRU except that documents that are larger than a given threshold are never cached. This policy tries to prevent the replacement of several small documents with a large document by enforcing a maximum size on all cached documents. Moreover, it implicitly assumes that a user tends not to re-access documents greater than a certain size. This is particularly true for users with low-bandwidth connections. LRU-MIN gives preference to small-size documents to stay in the cache. This policy tries to minimize the number of replaced documents, but in a way that is less discriminating against large documents. In other words, large documents can stay in the cache when replacement is required as long as they are smaller than the incoming one. If an incoming document with size S does not fit in the cache, the policy considers documents whose sizes are no less than S for eviction using the LRU policy. If there is no document with such size, the process is repeated for documents whose sizes are at least $\frac{S}{2}$, then documents whose sizes are at least $\frac{S}{4}$, and so on. Effectively,

LRU-MIN uses $\lfloor log_2(DocumentSize) \rfloor$ as its primary key and the time since last access as the secondary key, in the sense that the cache is partitioned into several size ranges and document removal starts from the group with the largest size range. The difference between LOG2-SIZE and LRU-MIN is that cache partitioning in LRU-MIN depends on the incoming document size and LOG2-SIZE tends to discard larger documents more often than LRUMIN. Hyper-G [6] is an extension of the LFU policy, where ties are broken according to the last access time. Note that under the LFU policy, ties are very likely to happen.

The Least Frequent Recently Used (LFRU) [8] cache replacement scheme combines the benefits of LFU and LRU schemes. In LFRU, the cache is divided into two partitions called privileged and unprivileged partitions. The privileged partition can be defined as a protected partition. If content is highly popular it is pushed into privileged partition. If it is require replacing content from privileged partition, the replacement is done as follows: LFRU evicts content from unprivileged partition, push content from privileged partition to unprivileged partition, and finally insert new content in privileged partition. In the above procedure, the LRU is used for the privileged partitions and approximated LFU (ALFU) scheme is used for the unprivileged partition; hence together is called LFRU. The basic idea is to filter out the locally popular contents with ALFU scheme and push the popular contents to one of the privileged partition.

B. Function-based Algorithms

Similar to key-based algorithms, function-based algorithms also rank cache objects but using a synthetic key. A synthetic key is a quantity associated with each cache entry and is calculated by combining multiple keys using a cost function. Usually, the keys have different weights in the cost function so that keys can be combined in the most balanced way. All function-based policies aim at retaining the most valuable documents in the caches, but may differ in the way they define the cost function. Weights given to different keys are based on their relative importance and the optimized performance metric. Since the relative importance of these keys can vary from one web stream of requests to another or even within the same stream, some policies adjust the weights dynamically to achieve the best performance. The GreedyDual algorithms [9] constitute a broad class of algorithms that include a generalization of LRU (GreedyDual-LRU). GreedyDual-LRU is concerned with the case in which different costs are associated with fetching documents from their servers. Several function-based policies are designed based on GreedyDual-LRU. They include the *Greedy Dual Size (GDS)* [10], the *Popularity-Aware Greedy DualSize (PGDS)* [11], and the *Greedy Dual* (GD*)* [12] policies.

Other function-based policies are based on classical algorithms (e.g., LRU). These policies include the Size-adjusted LRU (SLRU) policy [13]. The basic idea of Size-adjusted LRU (SLRU) is to orders the object by ratio of cost to size and choose objects with the best cost-to-size ratio. Least Relative Value (LRV) [14] assigns a value $V(p)$ for each document p. Initially, $V(p)$ is set to $\frac{Cp \times Pr(p)}{G(p)}$, where $Pr(p)$ is the probability that document p will be accessed again in the future starting from the current replacement time and $G(p)$ is a quantity that reflects the gain obtained from evicting document p from the cache ($G(p)$ is related to the size $S(p)$). As a result of this choice, the value of any document is weighted by its access probability, meaning that a valuable document (from the cache point of view) that is unlikely to be re-accessed is actually not valuable.

C. Semantic-aware Algorithms

SEMALRU [15] algorithm introduces a cache replacement policy based on document semantics, i.e. based on the content of the document. It is referred to as Semantic and Least Recently Used (SEMALRU). This algorithm assumes that for a period of time, the user seeks objects that are related to a given subject, or semantically close. This leads to a mechanism that least related objects to a new entry with respect to the semantics are preferred for eviction. In other worlds, SEMALRU tends to keeps objects which are closely related to specific user interests and discard documents which might be of less interest to users. The semantic closeness in SEMALRU is quantified by a parameter called semantic distance. This parameter is computed for every object in the cache. The documents with highest semantic distances is marked for eviction from cache. It is claimed that this semantic-based algorithm outperforms LRU.

There are several problems related to SEMALRU. First, the authors did not define an efficient way to calculate semantic distance of two pages. It assumes that semantic distances are calculated from webpage contents. Since scanning webpage contents to calculate similarity is rather expensive, computing the semantic distance for every object in cache consumes significant amount of computation, and thus, prevents the mechanism to scale.

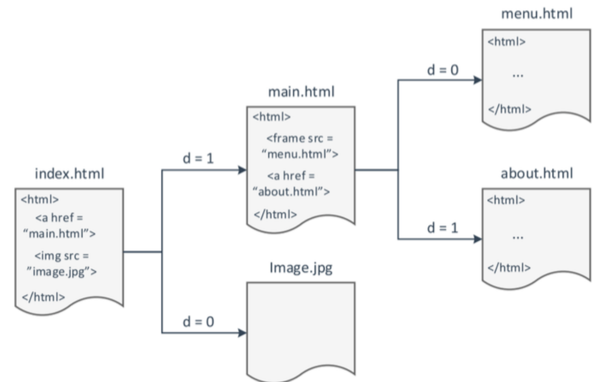

Fig. 1. Distance between objects

Lately in [2], Negrao et al proposed a semantic-aware cache replacement algorithm, called SACS, with the idea of adding a spatial dimension to the cache replacement process. Their solution is motivated by the observation that users typically

browse the Web by successively following the links on the web pages they visit. This reference relationship between pages via links in webpage content is a kind of semantic relations. SACS measures such semantic relations between pages by calculating the distance between pages in terms of the number of links necessary to navigate from one page to another.

Figure 1 shows a simple example of a web site and the corresponding distances between its pages. Let us denote $d(x, y)$ semantic distance between page x and page y. In this example, we have distance $d(\text{"}menu.html'', \text{"}index.html'') = 1$, and $d(\text{"}about.html'', \text{"}index.html'') = 2$.

At any moment, SACS keeps a list of recently accessed pages as pivots. For example, pages that are accessed within last α seconds. The distance $D(x_j)$ assigned to a page x_j is distance to the closest pivot p in the pivot set P. When replacement takes place, pages with smaller $D(x_j)$ are less probable to be evicted. It implies that pages far from the most recently accessed pages are candidates for removal. And the closer a page is to recently accessed pages, the less likely it is to be evicted.

Although SACS provides good caching performance to Web systems, it has several drawbacks. Firstly, at the beginning when no pages are in cache, pivot-base cache eviction does not function properly. Also the caching eviction performance is biased by what pages are populated in the cache first. Secondly, choosing pivots only by their recency may not be sufficient. It could be better if pivots are selected according to both their freshness and access frequencies. Finally, choosing all pivots that are accessed within the last α seconds could lead to high number of pivots for busy sites. A large number of pivots could degrade calculation of distances.

III. The Proposed Scheme

Our proposed algorithm, called Function-based Semantic-aware caching Alogrithm (or FSA), adopts the same concept of semantic distance as one used in SACS. In FSA, a pivot set $P = \{p_i | p_i \text{is a page in cache}\}$ is selected. During normal operations of the caching algorithm, a distance value $D(x_j)$ to pivot set P is calculated for each page x_j in the cache. Specifically, $D(x_j) = \min_{p_i \in P} d(x_j, p_i)$, where $d(x_j, p_i)$ indicates distance between page x_j and pivot p_i, which is the number of steps taken to navigate to x_j from p_i via reference links in page contents. Also cache entries are sorted according to their pivot distance value. Such a sorted list of cache entries facilitates finding eviction candidates when the cache needs more space for accommodating new Web pages.

A. Pivot Selection

Although FSA is the same as SACS in using navigating distance to pivot set for ranking cached pages, key differences that help improving FSA performance are the mechanisms used for refining pivot set

⋆ *Initial Pivot*: At the beginning, when there are no pages in the cache memory, a set of important pages are proactively chosen as pivots and put into the cache. Special pages of a site such as homepage, about page, category pages, etc. can be selected for such pages. By selecting initial pivots, cold-start situation happen in SACS is avoided. Also, initial pivots help cache administrators in better fine-tuning access pattern of users to the site by prioritizing some special pages that they want to promote.

⋆ *Pivot Value*: After the cache memory is populated with pages and objects, unlike SACS does, FSA does not select all pages that are accessed within the last α seconds for pivoting. Instead, only a subset of those pages are picked according to a quantity called *Pivot Value* (PV), which is calculated as

$$PV_i = N_i \times F_i$$

Here, N_i is order of page, which is defined as the number of links that can be accessed directly via the page i, and F_i is access frequency of that page. The higher PV_i value means the higher probability page i is chosen as a pivot. By building pivots from PV values, our pivot selection algorithm is superior to SACS in the following aspects. Firstly, it considers not only the recency or freshness of cached items but also access frequencies and content of cached items. Intuitively, a page that have many links to other pages and have high access frequency should be more preferred to be in cache and thus is a good candidate for picking as a pivot. Secondly, our mechanism of choosing pivot pages can keep the size of pivot pages small without degrading its quality since only a limited number of top pages are remained in cache.

B. Tie-breaking

In eviction phase, where pages are chosen to be removed from cache, tie can happen when eviction candidates have equal distances to pivots. Such cases happen frequently and require an additional cache value assign to each page to break tie. We proposed *Cache Value* CV that are calculated for candidate victims as a tie-breaking criteria:

$$CV_i = \frac{F_i}{C + S_i}$$

where, F_i is access frequency of victim i, S_i is size of i, and C is a constant that regulates the importance between access frequency and cache item size. Between two victims with the same distance to pivots, one that have higher CV value wins and remains in the cache. The other is evicted from cache to provide space for new caching items. By calculating CV as above, pages or other web objects that are frequently accessed and have small sizes are preferred to be in cache. The large value of C makes object size S_i less important than access frequency F_i

C. Complexity Considerations

In order to efficiently maintain up-to-date pivot sets as well as distance values and cache values for cached entries, FSA works in a reactive fashion as follows. When a request arrives to the cache, if it is a cache hit, access frequency of the corresponding cache item is updated and pivot set update could be triggered if necessary. If it is a cache miss, a request is made to upstream server which results in a new Web object to be

put into the cache. In this case, only distance value and cache value of the newly added object need to be calculated. After the new entry has distance value and cache value updated and it is stored in cache at its correct ordered position, eviction of cache entries could happen in case the cache is saturated. Such a reactive algorithm helps amortizing computation cost into request handling cycles and the cache data structure can be built incrementally. Particularly, for a cache hit request that does not change pivot set, computation cost is $O(1)$ (independent of total cache population) since a single cache entry need to update pivot value. For a cache miss request, the cost is proportional to the size of pivot set since distance values to each item of the pivot set to find the minimum value.

The worst scenario happens only when there is a cache hit that changes pivot set. In that case, all items in the cache need to update their own distance values to a new pivot set. Note that when this situation happens, only a single entry in pivot set is changed. This leads to computation cost of $O(N)$, where N is the total cache population.

Although changing in pivot set are costly and can degrade caching performance, it happens at much lower rate in comparison to SACS. Pages that are selected as pivots should have high pivot values, which depends on the order N_i of the page and its access frequency F_i instead of cache recency. This makes pivot set of FSA much stable than that of SACS. Choosing a good initial pivots can also improve efficiency of the caching mechanism even after the proxy cache server is started.

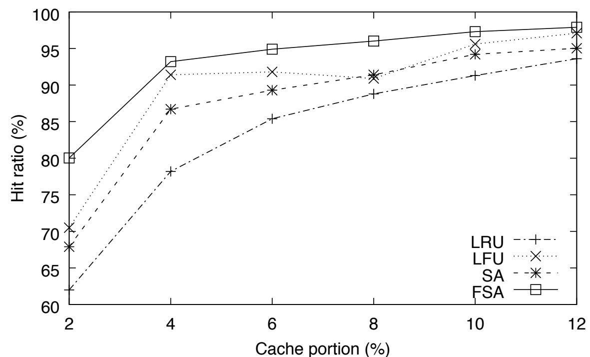

Fig. 2. Hit ratios on 1998-May-1

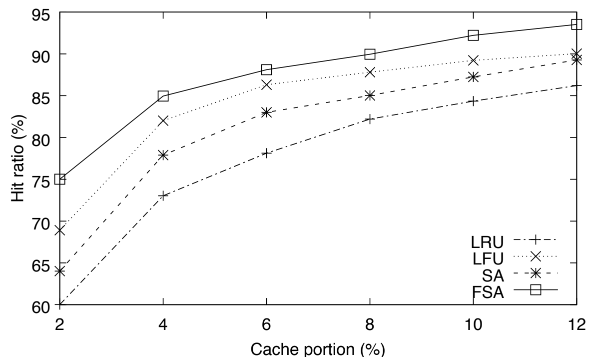

Fig. 3. Hit ratios on 1998-July-26

IV. Performance Evaluation

Our performance evaluation is performed using the access log of the FIFA World Cup 1998 web site [16]. The logs contain information about approximately 1.35 billion user requests made over a period of around 3 months, starting one month before the beginning of the world cup and finishing a few weeks after the end of this event. Each log entry contains information about a single user request, including the identification of the user that made the request (abstracted as a unique numeric identifier for privacy reasons), the id of the requested object (also a unique identifier), the timestamp of the request, the number of bytes of the requested object and the HTTP code of the response sent to user. We choose FIFA World Cup 1998 log trace dataset because it provides us all the information required to evaluate our algorithm and available for research purposes.

Our cache simulator is fully implemented using Java. Although it does not handle actual users HTTP request, its functionality is to measure hit ratio and byte ratio of the cache policy used. Hit rate measures the percentage of requests that are serviced from the cache (i.e., requests for pages that are cached). Byte hit rate measures the amount of data (in bytes) served from the cache as a percentage of the total number of bytes requested. These metrics are among the most commonly used to evaluate caching, and allow us to analyze the ability of our caching system.

More specifically, we build a server simulator that serves the HTTP requests in the access log data. Instead of returning response with payload data, the simulator returns nothing but a Boolean variable which indicate if the object is available in the cache. By that, the simulator is aware of the miss/hit status of each request, hence measure our metrics: hit ratio and byte ratio. On top of the simulator, we implement four different policies includes: LRU, LFU, SA (or SACS) and our policy FSA. We intend to compare our algorithm with traditional LRU and LFU, which are good overall algorithm commonly used in practice. We also compare with the original SACS algorithm to see if our improvement enhances the proficiency of the cache in our scenarios. The configuration parameters for our simulations are given in table II. We run the simulation of two different days: 1998-May-1 and 1998-July-26 which are respectively the very first day and last day of the league. Log file for each day contains around 1 million requests made to the website. For each caching algorithm we implemented, we recorded *hit ratios* and *byte hit ratios* and represented in the same plot for comparison purposes. In our simulations, *hit ratio* and *byte hit ratios* are defined as the fraction between the number of objects/bytes served from cache and the total number of objects/bytes requested

As we can see from plots in figures 2, 3, 4, and 5, FSA outperforms LRU, LFU and SA in term of hit ratio in both scenarios. It shows the efficiency of our proposed mechanisms

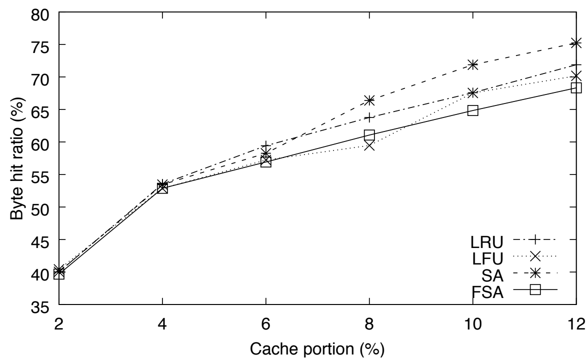

Fig. 5. Byte hit ratios on 1998-July-26

TABLE II
CONFIGURATION PARAMETERS FOR ALGORITHMS AND SIMULATIONS

Parameter	**Value**
Total data size	$\sim 100MB$
Cache size	from 2% to 12% of total data size
Number of web objects	~ 6500
Period for pivot selection (α)	$2s$
Pivot size limit	5
Constant C in cache value calculation	100

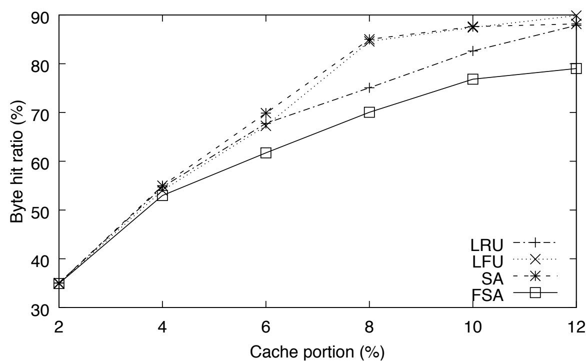

Fig. 4. Byte hit ratios on 1998-May-1

to previous mechanisms. However, when we consider byte hit ratios, it seems FSA does not bring differences in performance in comparisons to other algorithms. It is because FSA takes object size into account and favors small objects over large objects. As a result, there are more cache hits with small objects than large objects. This leads to very little improvement in terms of byte hit ratios.

V. CONCLUSION

In this paper, we have proposed an algorithm that combines a function-based and semantic-aware approaches in caching algorithm for Web systems. Our algorithm, called FSA, adopts the idea of web object distance from SACS and combine them with several parameters including recency, access frequency and object size in cache replacement cost function. FSA also provides some parameters that can be set by cache administrators in order to tune the cache system in different use cases. In our evaluation scenarios, FSA has the best performance in terms of hit ratio compared to existing algorithms. It also shows relative good result in terms of byte hit ratio.

REFERENCES

[1] W. Ali, S. M. Shamsuddin, A. S. Ismail, A survey of web caching and prefetching, Int. J. Advance. Soft Comput. Appl 3 (1) (2011) 18–44.
[2] A. P. Negrão, C. Roque, P. Ferreira, L. Veiga, An adaptive semantics-aware replacement algorithm for web caching, Journal of Internet Services and Applications 6 (1) (2015) 4.
[3] T. Koskela, J. Heikkonen, K. Kaski, Web cache optimization with nonlinear model using object features, Computer Networks 43 (6) (2003) 805–817.
[4] J. Cobb, H. ElAarag, Web proxy cache replacement scheme based on back-propagation neural network, Journal of Systems and Software 81 (9) (2008) 1539–1558.
[5] R. Ayani, Y. M. Teo, Y. S. Ng, Cache pollution in web proxy servers, in: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, IEEE, 2003, pp. 7–pp.
[6] A. Balamash, M. Krunz, An overview of web caching replacement algorithms, IEEE Communications Surveys & Tutorials 6 (2).
[7] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, S. Williams, Removal policies in network caches for world-wide web documents, SIGCOMM Comput. Commun. Rev. 26 (4) (1996) 293–305.
[8] M. Bilal, S.-G. Kang, A cache management scheme for efficient content eviction and replication in cache networks, IEEE Access 5 (2017) 1692–1701.
[9] N. Young, Thek-server dual and loose competitiveness for paging, Algorithmica 11 (6) (1994) 525–541.
[10] P. Cao, S. Irani, Cost-aware www proxy caching algorithms., in: Usenix symposium on internet technologies and systems, Vol. 12, 1997, pp. 193–206.
[11] S. Jin, A. Bestavros, Popularity-aware greedy dual-size web proxy caching algorithms, in: Distributed computing systems, 2000. Proceedings. 20th international conference on, IEEE, 2000, pp. 254–261.
[12] S. Jin, A. Bestavros, Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams, Comput. Commun. 24 (2) (2001) 174–183.
[13] C. Aggarwal, J. L. Wolf, P. S. Yu, Caching on the world wide web, IEEE Transactions on Knowledge and data Engineering 11 (1) (1999) 94–107.
[14] L. Rizzo, L. Vicisano, Replacement policies for a proxy cache, IEEE/ACM Transactions on networking 8 (2) (2000) 158–170.
[15] K. Geetha, N. A. Gounden, S. Monikandan, Semalru: An implementation of modified web cache replacement algorithm, in: Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on, IEEE, 2009, pp. 1406–1410.
[16] Worldcup 98.
URL http://ita.ee.lbl.gov/html/contrib/WorldCup.html