Negrão *et al. Journal of Internet Services and Applications* (2015) 6:4
DOI 10.1186/s13174-015-0018-4

Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

An adaptive semantics-aware replacement algorithm for web caching

André Pessoa Negrão[1,2*], Carlos Roque[1,2], Paulo Ferreira[1,2] and Luís Veiga[1,2]

Abstract

As the web expands its overwhelming presence in our daily lives, the pressure to improve the performance of web servers increases. An essential optimization technique that enables Internet-scale web servers to service clients more efficiently and with lower resource demands consists in caching requested web objects on intermediate cache servers. At the core of the cache server operation is the replacement algorithm, which is in charge of selecting, according to a cache replacement policy, the cached pages that should be removed in order to make space for new pages. Traditional replacement policies used in practice take advantage of temporal reference locality by removing the least recently/frequently requested pages from the cache. In this paper we propose a new solution that adds a spatial dimension to the cache replacement process. Our solution is motivated by the observation that users typically browse the Web by successively following the links on the web pages they visit. Our system, called SACS, measures the distance between objects in terms of the number of links necessary to navigate from one object to another. Then, when replacement takes place, objects that are distant from the most recently accessed pages are candidates for removal; the closest an object is to a recently accessed page, the less likely it is to be evicted. We have implemented a cache server using SACS and evaluated our solution against other cache replacement strategies. In this paper we present the details of the design and implementation of SACS and discuss the evaluation results obtained.

Keywords: WWW; Web performance; Web caching; Replacement algorithms; Semantic awareness

1 Introduction

Two decades after its inception, the World Wide Web continues to be among the most popular Internet services [1,2]. Everyday, an extensive number of users from all over the world accesses the Web to read online newspapers, get in touch with their friends on Social Networks, share their ideas on blogs or even play on video game websites. Information that used to be difficult to obtain is now one click away from us, on our laptops, tablets or even smart phones.

While the ubiquity of the Internet is desired by both users and content providers, the resulting high number of user requests is a challenge to the performance and scalability of Web servers and Internet Service Providers (ISPs) alike [3]. This challenge takes the form of, for example, increased computational load and bandwidth requirements due to the necessity of processing requests from (and delivering responses to) an increasingly larger number of users. This may, in turn, lead to increased latency, which results in user perceived delays [4].

In order to continue satisfying users' demands and, at the same time, reduce the network and computational strain imposed on content providers, several optimization techniques have been proposed over the years [5]. One of the most successful of such techniques is Web caching [6]. Web caching consists in storing copies of the pages requested by users to a web server on intermediary machines that can service future requests to those pages on behalf of the actual web server. Web caches optimize the operation of a web system by 1) reducing the load and network bandwidth at the origin Web servers by minimizing the number of accesses to the actual website and 2) minimizing access latency by placing data closer to the users [7]. For these reasons, web caches have become an ubiquitous background presence on the web and can be found at different layers of the Internet hierarchy [8].

Due to their limited sizes, caches cannot store every web page/object indefinitely. When the cache becomes full, the

*Correspondence: andre.pessoa@ist.utl.pt
[1]Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol 9, Lisboa, Portugal
[2]Distributed Systems Group, INESC-ID, Rua Alves Redol 9, Lisboa, Portugal

Springer

© 2015 Negrão et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

cache server must choose one or more objects to remove in order to make space for new objects. This decision is the responsibility of the cache replacement algorithm [9], one of the most critical components of a web cache system.

Among the most well known and highly used replacement algorithms are *Least Recently Used* (LRU) and *Least Frequently Used* (LFU), which remove the least recently or least frequently used object from the cache, respectively [10]. At the other end of the spectrum, more complex solutions based on machine learning try to predict which pages are going to be more frequently accessed by users based on past behavior [11-13]. In between, several other cache replacement algorithms have been proposed [14-21].

The most common approaches used in practice base their object replacement decisions on temporal locality information. In this paper, we propose adding a spatial dimension to the cache replacement problem. We propose a novel cache replacement algorithm, called SACS (from *Semantics Aware Caching System*), that takes into account semantic information about the cached web objects in order to decide which should be removed from the cache. More specifically, we propose looking into the structure of web pages, in particular the links between them, in order to try to predict which pages will be accessed in the near future. Our solution is based on the intuitive notion that an object that is linked by a recently accessed page has a higher probability of being accessed in the near future and, as such, should also be kept in the cache. Our system materializes this observation by assigning priorities to cached objects based on their link navigation information. This assignment is made in such a way that objects whose *distance* to recently accessed pages (measured as the length of the shortest path of links between them) is high have higher probability of being removed, while pages closer to recently accessed pages are kept cached. In addition, pages with the same distance factor are ordered according to their frequency information.

The main strength of our cache replacement algorithm is that it combines recency and frequency information with object access prediction based on the link relations between the different web pages/objects. By taking link information into account, SACS has additional information that allows it to reason in a more semantic way about future requests to the web cache server. This is in contrast with existing algorithms, which base their decision on syntactic information. In addition, by combining recency and frequency, our system is able to obtain the good results of these solutions in the scenarios in which they perform well, while, at the same time, being able to avoid their main shortcomings (e.g., cache pollution in LFU and eviction of popular pages that have not been recently requested in LRU).

The rest of this document is structured as follows. In Section 2 we describe the architecture of SACS and the design of the algorithm. In Section 3 we provide information about the implementation of our system. In Section 4 we present and discuss the results obtained in the evaluation of SACS. In Section 5 we discuss related work. Finally, in Section 6 we conclude the paper.

2 Architecture

SACS follows the execution loop of traditional cache management systems. When the cache receives a request for an object, it first verifies whether the object is already cached. If there is a version of the object available in the cache, that version is sent to the user. Otherwise, it is necessary to fetch the object from the origin web server and forward it to the user. In addition, the newly fetched document is placed in the cache. Doing so might require removing one or more objects from the cache, in case the available free space is not sufficient to hold the new object. The decision of which pages should be removed is made by the system's cache replacement algorithm.

In the next sections we present the design of SACS in detail. We start by introducing the main building blocks of our system in Section 2.1. Then, we describe SACS's cache replacement algorithm in Section 2.2.

2.1 Overview

Our system builds upon three main metrics: distance, recency and frequency. The *recency* metric tracks the last time an object was requested while in cache; it is the same metric used by the LRU replacement strategy. The *frequency* metric tracks the number of times an object was requested while in cache and is the metric used by the LFU replacement strategy.

The *distance* metric is a novel element introduced by our approach. It measures the distance between two objects in terms of the minimum number of links that need to be followed in order to navigate from one object to the other. In other words, given the web objects graph $G = (V, E)$, where V is the set of cached pages and E is the set of links between them, distance broadly corresponds to the length of the shortest navigation path between pages x_i and x_j. Implicitly, this metric is related with the probability that a user who requested object x_i will visit another object x_j to which there may be a navigation path rooted in x_i.

We denote the distance between two web objects x_i and x_j as d_{x_i,x_j}. Note that despite the notation implying otherwise, d_{x_i,x_j} refers to the distance of the navigation path from x_j to x_i; it means the *distance to get to* x_i *from* x_j. Due to the nature of the web, the distance function is not symmetric, which means that the distance between x_i and x_j might not be the same as the distance between x_j and x_i. In reality, it is clear that even if a page x_i is accessible from

x_j, the reverse might not be true, since there might not be a navigation path between x_j and x_i. It should also be clear that distance applies only when the root of the navigation path in consideration is an HTML page; the destination, on the other hand, can be any type of web object.

The distance between two objects depends not only on the number of links existing in the navigation path between the pages, but also on the type of HTML constructs (i.e., tags) that make up the links. We distinguish tags that must be explicitly clicked by the user (*explicit links*) from tags that refer to objects that are automatically loaded when the requested page is parsed (*implicit links*). Regarding explicit links, our system considers only the *a* HTML tag. A link in the navigation path referring to this tag is assigned a distance of 1, since traversing this link requires the user to follow the link, which is not guaranteed to happen.

As for implicit links, we consider tags *img*, *link*, and *frame*. Other tags, such as *script*, *audio*, *video* and similar ones, may also be considered, depending on the types of objects that the cache administrators consider to be cacheable. Implicit links, unlike explicit ones, are assigned a distance of 0, since they are aggregated with the object that is at the origin of the link and are meant to be loaded simultaneously. Figure 1 shows a simple example of a web site and the corresponding distances between its pages. In this example, we have $d_{menu.html,index.html} = 1$, while $d_{about.html,index.html} = 2$.

Consider the set V of direct links between cached pages. Further consider function τ that, given two objects, returns the type (implicit or explicit) of the link between the two. The following equation formalizes the definition of *link distance*:

$$d_{x_i,x_j} = \begin{cases} 0 & \text{iff } (x_i,x_j) \in V \ \wedge \ \tau(x_i,x_j) = \text{implicit} \\ 1 & \text{iff } (x_i,x_j) \in V \ \wedge \ \tau(x_i,x_j) = \text{explicit} \\ \infty & \text{iff } (x_i,x_j) \notin V \end{cases} \tag{1}$$

The following section explains how our system measures the distance metric and how it is used, along with the other metrics, by our cache replacement algorithm.

2.2 Cache replacement

At any moment, SACS keeps track of a set of pages that have a special interest to our replacement algorithm. We refer to these special pages as *pivots* and denote the pivot list as P. The importance of pivots stems from the fact that it is with relation to them that the distance assigned to each cached page is determined. More specifically, the distance assigned to a page x_i corresponds to its distance to the closest pivot:

$$d_{x_i} = min\left(d_{x_i,p_1}, \ldots, d_{x_i,p_n}\right), \forall p_i \in P \tag{2}$$

If the object in question is directly linked by a pivot (i.e., $\exists p_i \in P : (x_i,p_i) \in V$) or if it is not linked by any other page (i.e., $\neg\exists x_j : (x_i,x_j) \in V$), then its distance is given by Equation 1. Otherwise, if the object has links to it, but none is a pivot, its distance is defined recursively as follows:

$$d_{x_i,p_i} = d_{x_i,x_j} + d_{x_j,p_i}, \forall x_j \in Parents(x_i) \tag{3}$$

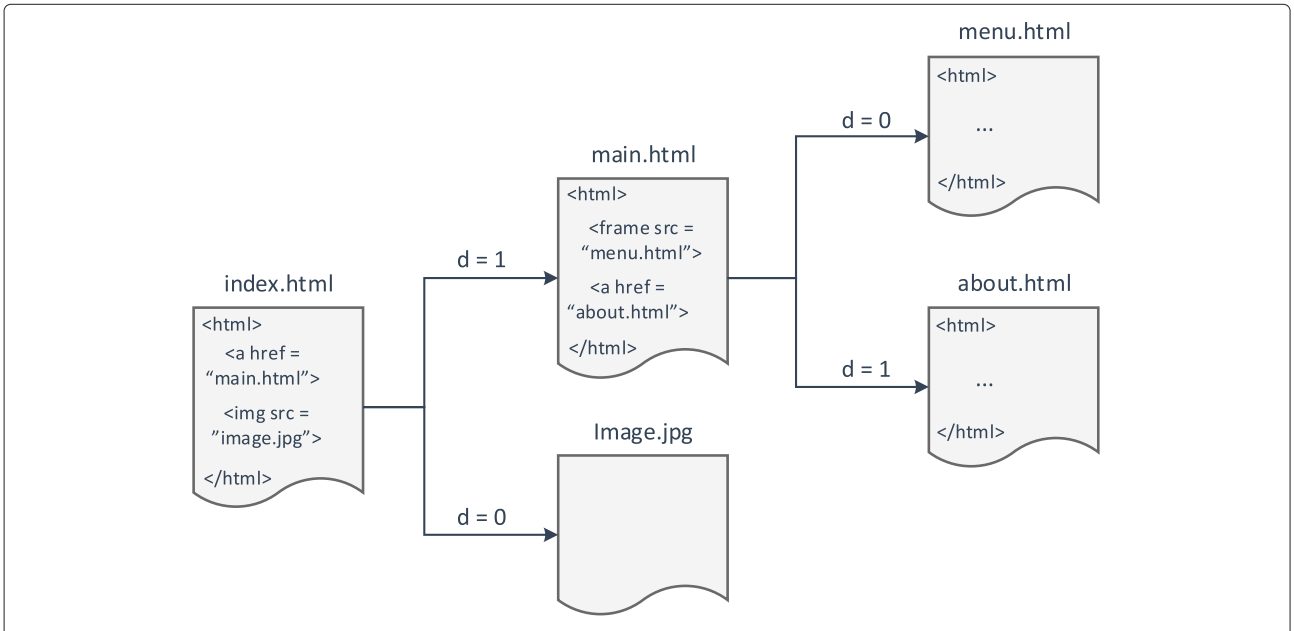

Figure 1 Example of a web site with distance assigned based on link information.

In our system, pivots correspond to the most recently accessed cached pages; it is this feature that allows us to prioritize caching the most recently accessed pages, similarly to LRU. More specifically, pivots correspond to the cached objects that have been requested in the last α seconds, where α is a parameter set by cache administrators. Consider that t is the current time and t_{x_i} is the timestamp of the latest request to object x_i. Further consider that R is the list of past requests submitted to the cache. The definition of *pivot* is formalized in the following formula:

$$x_i \in P \iff x_i \in R \wedge t - t_{x_i} < \alpha \quad (4)$$

Algorithm 1 Processing received requests

```
1:  function RECEIVEREQUEST(URL, requestCount)
2:      if URL ∈ cache then
3:          obj ← cache[URL]
4:          obj_timestamp ← requestCount
5:          obj_count ← obj_count + 1
6:          pivots ← pivots ∪ URL
7:          response ← msg(obj_data)
8:      else
9:          obj ← GET_DATA_FROM_ORIGIN(URL)
10:         if obj is cacheable then
11:             if obj_size ≤ cache_maxSize − cache_size then
12:                 cache[URL] ← obj
13:             else
14:                 EVICT_AND_REPLACE(obj)
15:             end if
16:         end if
17:         response ← msg(obj_data)
18:     end if
19:     return response
20: end function
```

Our algorithm comprises two main phases: monitoring and eviction. In the monitoring phase (see Algorithm 1), the cache collects information about access to pages and user requests. The algorithm keeps, for each cached page, the page's hit count while cached (*frequency*) and the timestamp of the latest access to the page (*recency*). On a cache hit, the frequency and recency information of the requested object is updated and its cached version is returned to the user. If, on the other hand, the requested object is not cached, the system gets it from the origin server before returning it to the requesting user. In addition, a cache miss triggers the eviction phase of the algorithm if the requested object does not fit in the available free space.

The eviction phase (Algorithm 2) is triggered when the free space available in the cache is not sufficient to accommodate new pages. At that moment, the algorithm chooses one or more pages to remove until a predefined *target cache size* threshold is reached. Since there may be different layers of storage within the server (e.g., memory and disk), each area has its own set of thresholds for the maximum and target cache size, so that they can be customized and controlled independently by the cache server administrator. The threshold values are specified in bytes and respect the property $TargetCacheSize < MaxCacheSize$.

To choose which page(s) to evict, our algorithm sorts the candidates for replacement according to our replacement policy. For performance reasons, the candidates for eviction consist of a randomly selected sample of size β of the cached objects. β can be freely set by the deployers of the cache according to their needs. We analyze the impact of this parameter in the performance of SACS in Section 4.3.3.

Our eviction algorithm requires obtaining the distance of each eviction candidate. Because computing this distance requires executing a search algorithm, it is not feasible nor scalable to execute this algorithm on the fly for every candidate object every time eviction is issued. Instead, our system continuously executes the distance computation function in the background. As a consequence, every object has its distance refreshed periodically. Our distance computation algorithm executes a Breadth First Search with bounded depth over the page graph. We bound the maximum depth of the search not only for performance reasons, but also because after a certain distance, the probability of a page being visited by following the links in a path decreases sharply. The search returns the depth of the first pivot encountered or *maxDepth*, if it reaches the maximum depth without finding a pivot.

Algorithm 2 Eviction phase

```
1:  function EVICTANDREPLACE(obj)
2:      if obj_type is HTML then
3:          CRAWLER.PARSE(obj_data)
4:      end if
5:      sample ← PICK_RANDOM_SAMPLE(β)
6:      for c ∈ sample do
7:          if c ∈ pivots then
8:              c_distance ← 0
9:          end if
10:     end for
11:     i ← 0
12:     SORT(sample)
13:     while cache_size ≥ cache_targetSize do
14:         cache.REMOVE(sample[i])
15:         i ← i + 1
16:     end while
17: end function
```

With the distance computation running in the background, during the eviction phase our system simply retrieves the distance information from the metadata of each candidate page. We perform one additional step that verifies, for each candidate, if it hasn't become a pivot since its distance was last refreshed by the distance computation function. After the distance information is obtained, the eviction candidates are ordered by their distance, with objects with higher distance being ordered before objects with lower distance. Then, the ordering of objects with the same distance depends on the frequency information of the pages, such that objects less frequently accessed are ordered before pages more frequently accessed. Objects are then removed in order until the target cache size threshold is reached.

Finally, when the new object is added to the cache, SACS parses it in the background to extract its link information and updates the link information in the page graph. As for the pages that are evicted, the cache does not remove their information from the graph immediately, as it may still be useful to compute another page's distance. Instead, this information is garbage collected either when no other cached page links to the removed page, or when memory becomes scarce.

3 Implementation

We implemented a prototype of a web cache server that uses SACS as its cache management system. Our implementation was written in Java and is built on top of the *Ehcache* [22] open-source cache software. Ehcache is a general-purpose caching library that implements a generic object cache that can be used to implement different types of caching systems. Ehcache implements both memory and disk caching, coupled with a simple API for managing interaction between the different caching layers. It comes preloaded with three cache replacement algorithms for the in-memory cache (LRU, LFU and FIFO), but allows custom made replacement policies to be plugged into the system. For the disk cache, however, only the LFU replacement strategy is available.

Ehcache stores cached data as generic *Element* objects. An element contains a key, a value and a set of useful metadata, such as the element's hit count and the timestamp of its last access. To add our own functionality, we created a *SACS Element* extending the original Element class with additional metadata required by our replacement algorithm.

Plugging our cache replacement algorithm into Ehcache required us to implement a *SACS Policy* class extending Ehcache's *Policy* interface. The Policy interface exports only three methods, which are necessary for deciding which element to evict from the cache. The most relevant of these is method *selectedBasedOnPolicy*, which receives a list of candidate elements from which the element to remove is selected. However, when called by Ehcache, this method receives only a small sample of the cached elements, which limits the control of our system over the eviction process. As such, in order to grant our replacement algorithm access to the full element list, we had to make a few modifications to the source code of Ehcache. Further modifications to the source code of Ehcache had to be made in order to allow our system to control the value of the target cache size, which is not exported by Ehcache's configuration tools.

Besides Ehcache, our implementation uses two additional external libraries: *Netty*, and *htmlparser*. *htmlparser* [23] is a Java based html parsing library we use to parse HTML pages and extract link information from them. Netty [24] is a network I/O framework that abstracts different network protocols into a high level API, easing the effort of implementing network applications. We used it to handle several lower level aspects of the network communication part of the system, such as receiving and sending messages and managing user and server connections. We also used its HTTP parsing API to extract information from HTTP messages.

We keep a graph with the link information in store, indexed by a hash table, for direct access to the nodes in the graph. The graph is constructed iteratively, as new pages are added to the cache, and periodically by crawling the cached contents in the background. Crawling is necessary because parsing a page only obtains the links that originate in it, not the ones that point to it.

In our current implementation, we set the sample size to 10%, for performance and scalability reasons. To identify if an object is a pivot, we apply Equation 4. In other words, we verify if the timestamp of the last access to the object is within an interval of α seconds. In our current implementation, $\alpha = 120$.

4 Evaluation

We have conducted a series of trace-driven experiments with the goal of analyzing the performance of SACS regarding a number of metrics and execution scenarios. In this section, we provide detailed information about the evaluation process and the results obtained.

4.1 Dataset

Our evaluation was performed using the access logs of the FIFA World Cup 1998 web site [25]. 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 conclusion 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 the user.

The main reason for choosing the FIFA logs is the fact that it provides us, directly or indirectly, with all the information required to evaluate our algorithm: 1) it directly provides us with user driven access patterns that represent the real way users have requested pages from the site; 2) it provides us with sufficient information to obtain the actual site, through which we can obtain crucial information about pages and links; and 3) it is readily available from the Internet with a manageable size. Other traces found online or used in literature often omit important information (such as the requested URLs) by anonymizing the traces or are no longer available.

Although the dataset does not include an actual copy of the web site, the logs come with a file that maps the unique identifiers of the web objects to their respective URLs. Since our solution requires that we have access to the link information (which is only available within the web pages), we used the Internet Archive [26] to download the web site. However, the Internet Archive does not possess a full copy of the web site. As a result, we were only able to obtain approximately 70% of the html pages of the website. Instead of discarding the remaining pages from the experiments, we created stub replacements for them containing no links to other pages. This allowed us to simulate a more realistic scenario, since in a real world setting, not every page in the cache will have links to other cached pages.

For most of the evaluation scenarios, we focused on an eleven day period from June 24 (day 60) to July 4 (day 70). This period contains the busiest days in the logs, with an average of 40 million requests per day, with the highest daily request count peaking at 80 million on July 1 (day 67). In total, this interval includes close to 450 million request, one third of the total number of requests in the logs.

4.2 Simulation environment

The main performance metrics considered in this evaluation were *hit rate* and *byte hit rate*. 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 amount of bytes requested. These metrics are among the most commonly used to evaluate caching systems [9,11-13,27], and allow us to analyze the ability of our caching system in caching the pages that are most likely to be requested in the near future.

We compared the results obtained by our solution with LRU and LFU. Our implementation of LRU evicts from the cache the pages with the oldest request number, while LFU evicts the ones with the lowest hit count. LRU is a good overall algorithm that is commonly used in practice (for example, in Squid [28]). LFU is also a popular replacement algorithm that works well when the set of the most popular pages is mostly stable or changes slowly. However, it is subject to cache pollution when popularity changes more dynamically, as it tends to maintain in the cache objects that have been popular, but no longer are. Since the traces used in our experiments do not display this pattern of changes in popularity (object popularity in the logs is mostly stable), it does not allow us to clearly analyze how SACS compares with LFU. For this reason, we designed a synthetic scenario in which before running the simulation, we populate a percentage of the cache with pages that are not frequently requested in the logs. In addition, we set the initial hit count of those pages to a higher value than they would naturally have. In the evaluation, for each algorithm considered, we present the results obtained with this *biased* scenario against the *regular* scenario that starts with a clean cache.

In our simulation, uncacheable and invalid requests, which would have resulted in cache misses, are discarded and are not accounted for in the statistics. As a consequence, the values for both hit rate and byte hit rate obtained and presented here are higher than they would be in reality and can, thus, be regarded as an upper bound of the real values. However, since these requests would have been discarded by all algorithms, they only have a numeric impact on the results; they have no effect on the relative results of the algorithms evaluated and, as such, do not affect the reliability of the results obtained.

To analyze the performance of the system under different cache sizes, we executed each simulation with a cache size of 4MB and repeated it with an 8MB cache size. Although these values are small in absolute terms, they correspond, respectively, to 5% and 10% of the total size of the cacheable contents of the FIFA98 website.

Our experiences consisted in sequentially reading entries from a set of log files and feeding them directly to our cache server. As a result of this approach, issues related to concurrency are not considered, except later in Section 4.3.4. For each experiment we executed multiple simulations, logging the values obtained in each individual simulation and computing the corresponding averages. The values presented in this section correspond to the averages obtained. The experiments were executed on a single machine equipped with an Intel Core i7 870 2.93GHz CPU (4 cores) and 12GB of RAM running Ubuntu Linux.

4.3 Results

In this section we present the results obtained during the evaluation of our system. First, we present the results of our comparative analysis of SACS against LRU and LFU regarding hit and byte hit rate. We then compare the performance of SACS in terms of memory usage and

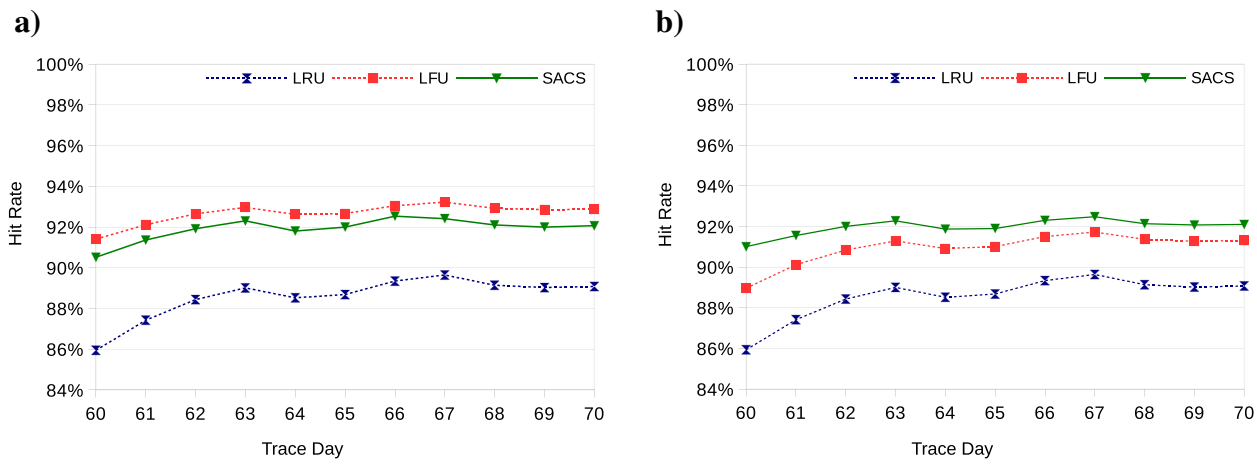

Figure 2 Hit rate with 5% cache size under different cache pollution scenarios. (a) Regular hit count. **(b)** Biased hit count.

throughput against the two competing algorithms. We follow this by analyzing the behavior of our system with different sample sizes. Finally, we evaluate the impact of network delays and concurrency on SACS.

4.3.1 Hit rate and byte hit rate

The first set of figures presented in this section compares the results obtained by the three systems under evaluation regarding hit rate. In the results presented in this section, we configured the sample size of all the algorithms to be the full cache (i.e., sample size = 100%). This allows us to compare the algorithms without perturbations from the inherent non determinism of the sampling process.

Figures 2 and 3 show the average hit rates obtained while executing several simulations with a cache size of 5% and 10%, respectively. Figure 2(a) shows the hit rate values obtained with a 5% cache size in the regular scenario (i.e., the cache starts empty). The results show that SACS is able to match the performance of LFU, falling just short of 0.5% of its performance, at most, while outperforming LRU by over 5%. LRU has a worst performance because frequently requested objects are not given higher priority over more recently accessed objects that are requested infrequently. This way, a popular object that, by chance, has not been accessed for some time may be removed by LRU instead of an object that, while recently requested, ends up being only accessed once. LFU, on the other hand, avoids this problem by keeping the most frequently accessed objects in the cache, regardless of the time of access. In the same way, SACS is also able to tackle this issue by weighing distance/recency with frequency. In particular, recently accessed objects that are no longer pivots, which with LRU would likely be kept, are only kept by SACS if their distance to a pivot and their frequency so determines.

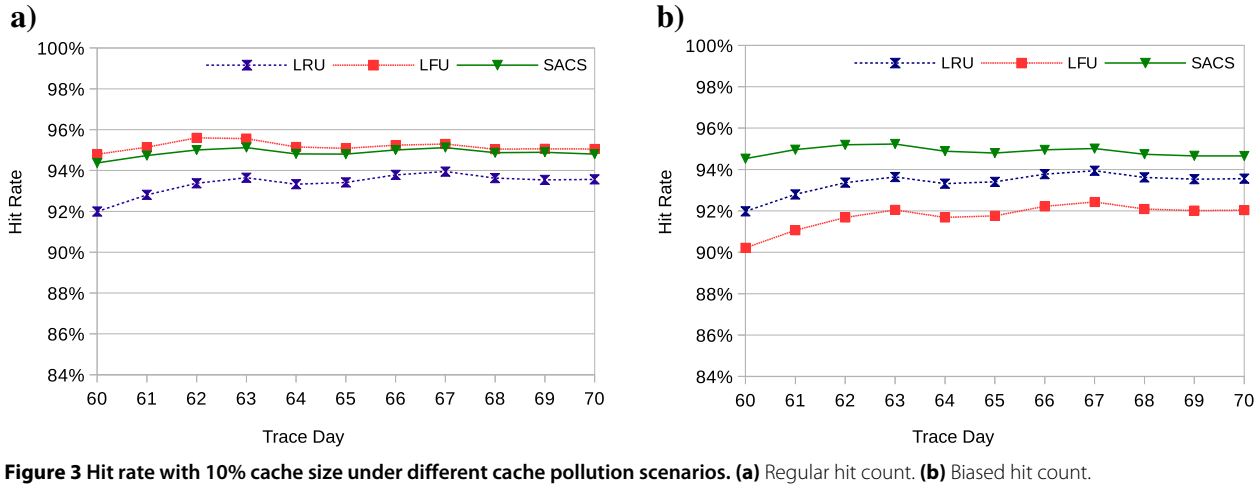

Figure 3 Hit rate with 10% cache size under different cache pollution scenarios. (a) Regular hit count. **(b)** Biased hit count.

Although LFU performs better than LRU in the previously analyzed data, the reason for its good performance is due to the characteristics of the data access patterns of the FIFA98 logs. Specifically, it is due to the fact that in these logs, the set of the most popular pages is stable and mostly fixed over time. However, it is well known that page popularity in the Internet varies dynamically and unpredictably. Thus, in order to analyze the performance of the algorithms under a more dynamic setting, we ran our biased scenario, which pre-populates the cache with a set of infrequently accessed pages. The results obtained in this scenario are shown in Figure 2(b).

As expected, the performance of LFU in the biased scenario is lower than in the regular scenario. This performance decay is due to the presence of objects that have high hit counts, but that are no longer frequently accessed. As the figure shows, this performance reduction is higher at the start of the simulation, which is when cache pollution is at its highest level. As the simulation moves forward, LFU's performance steadily increases. Once again, this steady improvement is a result of the characteristics of the FIFA98 logs, which generate a very low and unnoticeable rate of cache pollution.

SACS and LRU, on the other hand are less, if at all, affected by the cache pollution scenario, because both take recency in consideration. For LRU, frequency is completely ignored, which means that the polluting objects are readily removed from the cache and have a low impact on cache performance. And while SACS has frequency in consideration, even the pages with the highest hit count can become valid candidates for eviction if they are not close to a pivot. Hence, distance is able to filter frequently used pages that *are* relevant from frequently used pages that *have been* relevant, but no longer are.

The results presented thus far are further confirmed by Figure 3, which refers to the same scenarios analyzed before, but this time regarding a 10% cache size. As before, and for the same reasons, LFU and SACS obtain similar results in the regular scenario, and outperform LRU. The biased scenario also follows the same pattern, with SACS and LRU not being affected by the cache pollution, unlike LFU. The main differences between Figures 2 and 3 are 1) the hit rate values, which are, as expected, higher with the largest cache, and 2) the magnitude of the performance decay of LFU, which in the case of the 10% cache size falls below the performance of LRU.

Byte hit rates (shown in Figures 4 and 5) follow the same pattern as the hit rate, both for the different hit count scenarios and the different cache sizes. However, the exact values for byte hit rate are about 10% lower than hit rates. The reason for this discrepancy is that while every hit/miss has the same weight (1) regarding hit rate, they have have different weights regarding byte rates, which depend on the sizes of the objects to which the hit/miss refers. Because in the FIFA logs some of the least frequently used objects are among the largest ones, byte hit rate is lower, since these large objects have a higher weight and often lead to misses.

4.3.2 System performance

We now take a look at the behavior of SACS from the perspective of system performance, when compared with LRU and LFU, still considering full cache sampling. We focused on two main system performance metrics: memory consumption and throughput (in number of requests serviced per second). We gathered memory usage information from within the simulation code by accessing the *\proc* virtual filesystem of Linux. Throughput information was collected by dividing the total number of requests received at the server by the total execution time of the simulation (considering the simulation starts when the first request is received). Note that the values

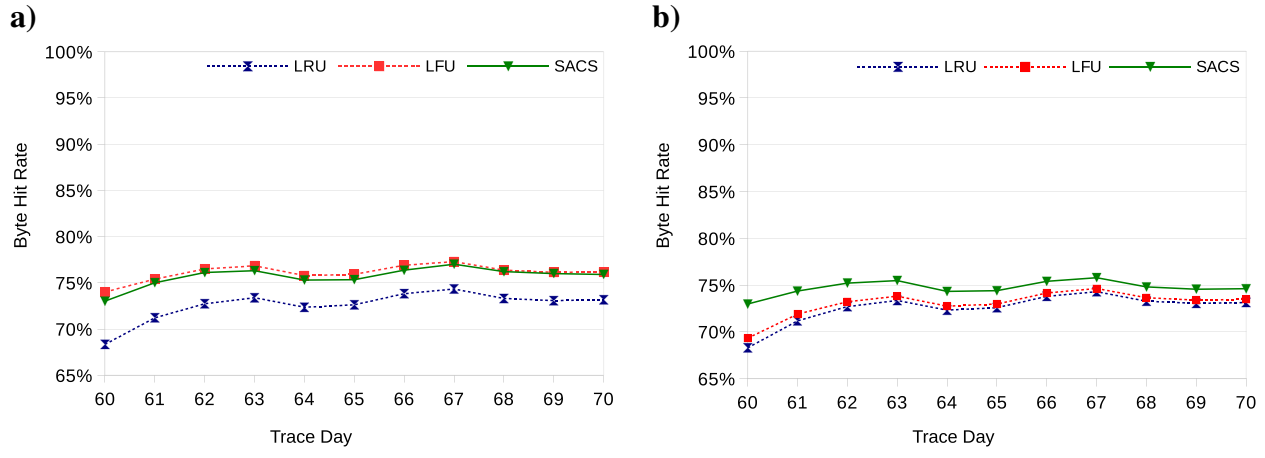

Figure 4 Byte hit rate with 5% cache size under different cache pollution scenarios. (a) Regular hit count. **(b)** Biased hit count.

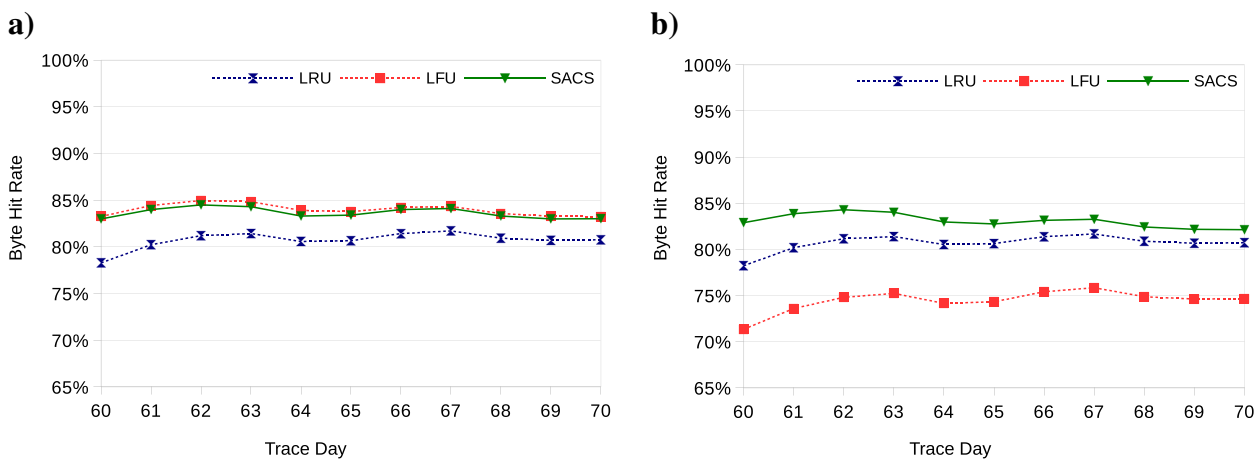

Figure 5 Byte hit rate with 10% cache size under different cache pollution scenarios. (a) Regular hit count. **(b)** Biased hit count.

obtained reflect only the processing time at the cache server, since no network communication occurs during the simulation.

Table 1 presents the memory usage results. The values presented correspond to the maximum amount of memory used by each strategy, excluding the memory reserved for the actual cache contents. This means that these results reflect only the additional data that each algorithm needs to maintain. For this same reason, the values measured were identical for both cache sizes used in the simulation.

As expected, the results show that our solution uses more memory than the competing strategies. This extra memory corresponds to the additional information and data structures used by SACS (in particular, the link/page graph) which are absent from LRU and LFU. Despite the overhead, the actual difference is only of 3MB, with an average of 700 graph nodes at any one time during the simulations. For comparison, column *SACS Max* shows the maximum possible amount of memory that would be used by SACS if the page graph contained information about every object (cached or not) seen by the cache server during the simulation.

Figure 6 presents the throughput achieved by the different algorithms with 5% and 10% cache size. Unlike memory usage, throughput varies with cache size, with larger caches delivering higher throughput because object replacement is issued less frequently. Similarly to memory usage, the figure indicates that SACS has a slight computational overhead when compared to the two alternatives. The reduced throughput obtained is a result of the more complex computations required to select the most appropriate object in SACS. The main reason for this overhead is, as expected, the distance computation function, which continuously executes in the background.

It should be noted, however, that due to the characteristics of the simulation, the delays associated with responding back to users and requesting missing pages from the origin server are not reflected in the results. We argue that if these delays are taken into account the throughput of our system would be competitive with that of LRU and LFU, for the following reasons. First, the overhead due to the background distance computation process, would be masked by the network delays of fetching the missing pages, which are an order of magnitude larger than computational delays. Second, because our solution is able to obtain an overall higher hit rate, it would issue requests for missing pages less often, which would, in practice, lead to a higher responsiveness of the caching system. We further analyze this claim later in Section 4.3.4.

Table 1 Total memory used

Replacement strategy	RAM used
LRU	8MB
LFU	8MB
SACS	11MB
SACS Max	20MB

4.3.3 Cache sampling

So far, we have analyzed the performance of the systems under evaluation in a stable, concurrency free environment that favors hit rates, but exposes computational overheads more clearly. The overheads identified so far are, for the most part, due to the fact that the simulations have been conducted using full cache sampling. This means that every cached object has been considered as a candidate for eviction, which requires the algorithms to have to analyze and evaluate every object in the cache before one or more of them is selected for removal.

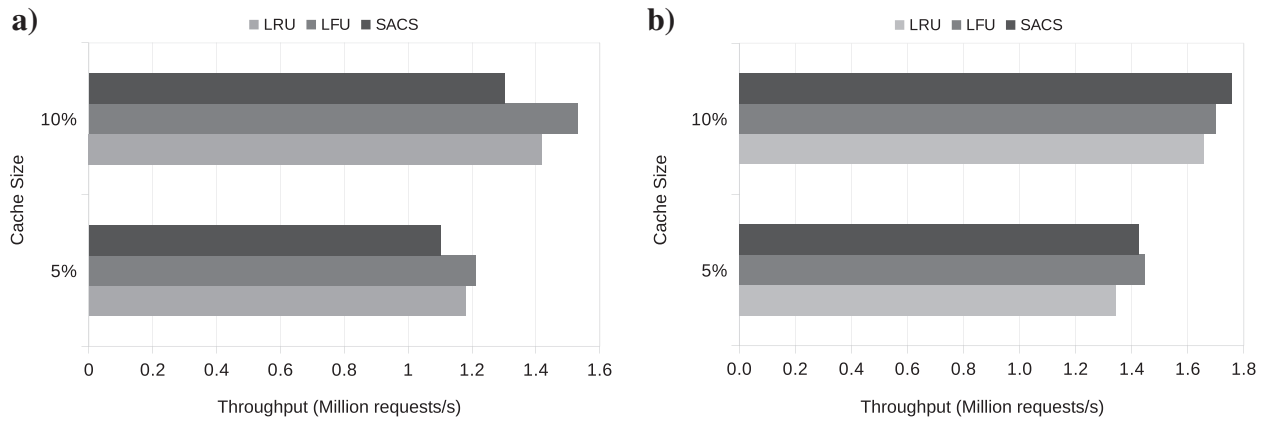

Figure 6 Throughput comparison for different cache sizes and cache pollution scenarios. (a) Regular hit count. **(b)** Biased hit count.

However, while this solution is able to maximize hit rate, it clearly is not scalable to larger caches in which the number of objects to analyze would be prohibitively high. The consequence is the reduction of throughput, which can lead to higher response times.

A strategy frequently used by commercial web caches (such as Squid) to circumvent this scalability limitation is to randomly select a sample of the cached objects as candidates for eviction, instead of the full cache. In this section, we analyze the impact of sampling in the performance of SACS. Based on Squid and Ehcache policies, we start by analyzing the performance of SACS with a sample size of 10% of the total cache size and compare it against LRU and LFU with the same sample size. After that, we analyze the performance of SACS alone with different sample sizes.

Figure 7 shows the results obtained by the three algorithms with a 10% sample size. Figure 7(a) presents the hit rates obtained with a 5% cache size, while Figure 7(b) presents throughput results for both 5% and 10% cache sizes. Both plots show the averages of the results obtained over a series of simulations of the *regular* (non biased) scenario. Unlike the previous figures (which showed results only for the interval between days 60 and 70), the results presented in Figure 7 are from a series of simulations spanning the full extent of the logs, starting in day 5 (the first with access information) until the last day.

The hit rates obtained during the simulations (byte hit rates, which we omit due to space limitations, follow an identical pattern, but with lower scores) show that SACS outperforms both LRU and LFU with a 10% sample. SACS tops the two competing algorithms in every day of the simulation with an average of 1.5% over LFU and 1.75% over LRU (with a maximum daily difference of 3.2% and 3.6%, respectively). Similar results were obtained with a 10% cache size, but with slightly higher overall hit rates

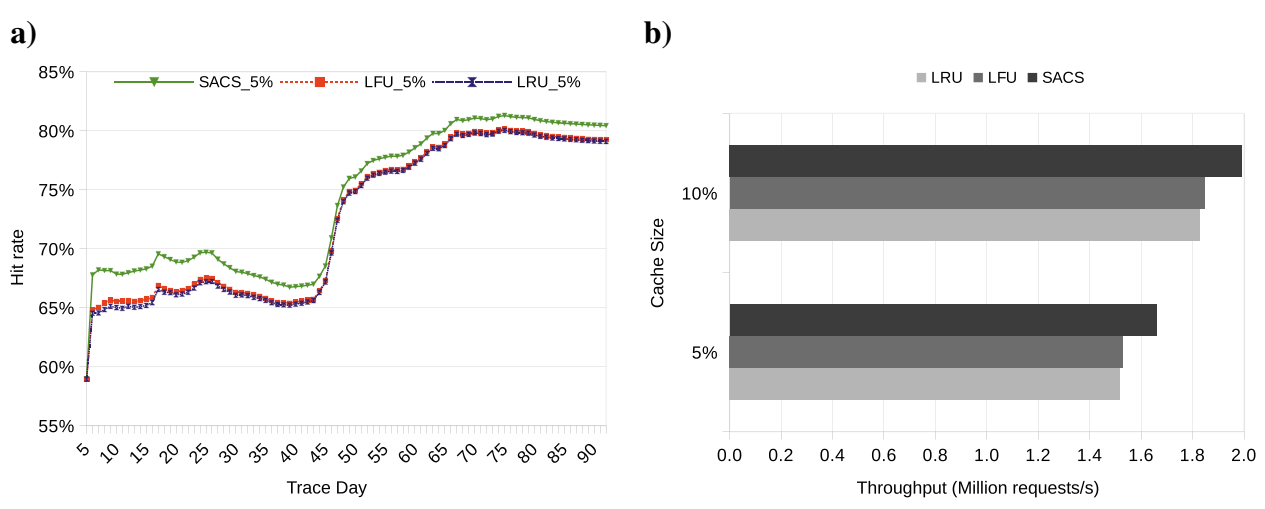

Figure 7 Hit rate and throughput with 10% sample size. (a) Hit rate per day. **(b)** Overall throughput.

for all the algorithms. The results confirm the ability of SACS to adapt to different situations and to obtain the most out of the information available to make replacement decisions. Overall, hit rates are 10% lower than the ones obtained with a 100% sample size, because sampling does not allow the algorithms to make a fully informed decision.

Contrary to hit rates, however, throughput is higher with smaller sample sizes. SACS, in particular, obtains a significant boost over the other two algorithms. This increase in throughput is a direct consequence of the lower complexity of the eviction algorithm, which now has to process a much more compact candidate list. As a consequence of the increased throughput, the cache will be able to respond more quickly to requests for cached objects. In contrast, a higher number of requests are not serviced from the cache, which means that the average response time may increase.

The results presented so far indicate the complex relation between hit rate and throughput, in particular regarding their combined effect on response times. On the one hand, higher hit rates inherently results in higher throughput, because more objects are serviced from the cache, bypassing the need to obtain the object from the origin web server. On the other hand, using larger samples to obtain higher hit rates results in lower throughput, due to the additional computational overhead of the eviction algorithm. To further illustrate this tradeoff between hit rate and throughput, we executed a series of simulations with SACS using different sample sizes and logged the average throughput and hit rates obtained over the various executions. The results are presented in Figures 8(a) and 8(b). The plots show the results obtained by SACS with a 20% and 30% sample size, accompanied, for comparison, by the results of the full (100%) and 10% samples already presented before.

The tradeoff between throughput and cache accuracy/performance is particularly evident if we compare the throughput and hit rate obtained by the SACS_100% and SACS_10% configurations. For these configurations, the results show, on the one hand, that SACS_100% obtains the highest hit rate, while SACS_10% obtains the lowest hit rates, with a difference of approximately 10% between the two. On the other hand, the throughput of SACS_100% is the lowest of all configurations, by a clear margin, with a 26% difference to SACS_10%. Reducing the sample size from 100% to 30% results in a 6% reduction in hit rate, but in over 15% increase in throughput; reducing to 20% (from 100%) results in a 8% loss in hit rate and a throughput increase of 23%.

4.3.4 *Network simulation*

We finalize our evaluation by analyzing our previous claim (last paragraph of Section 4.3.2.) that in a real setting with concurrent accesses to the cache and network delays, the overhead caused by SACS's background tasks would be less significant. To this end, we modified our simulator so that requests would be issued by multiple threads rather than sequentially by a single process. Requests from each individual user, however, are still issued sequentially, although concurrently with other users.

In addition to issuing requests concurrently, we introduced artificial delays on cache misses to simulate the process of fetching pages not present in the cache from their origin web server. The delays were based on offline measurements we conducted by issuing requests to web servers placed on different locations. Similarly to the previously described experiments, we executed several

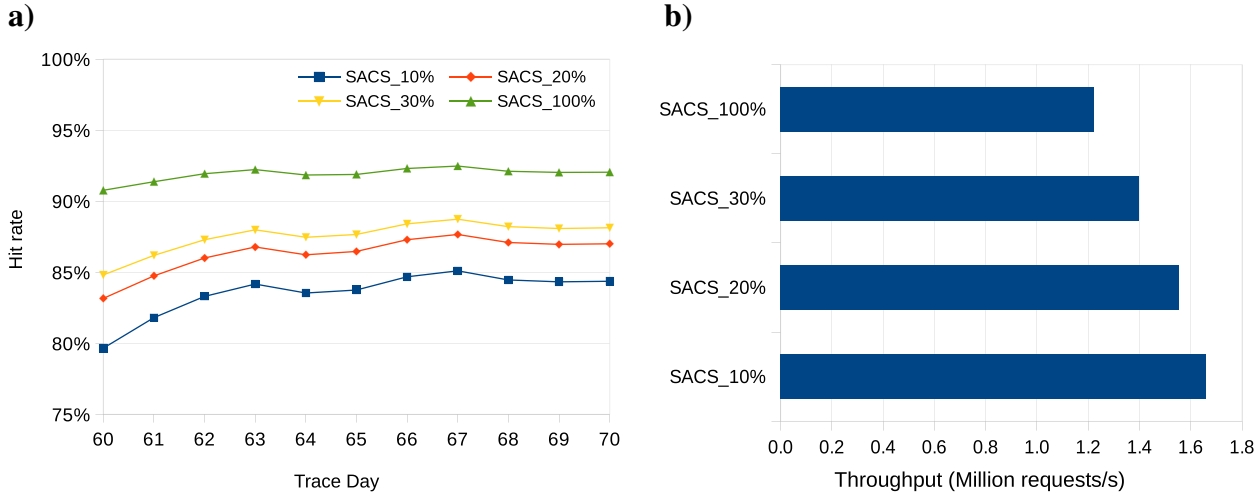

Figure 8 Hit rate and throughput for SACS with different sample sizes. (a) Hit rate per day. **(b)** Overall throughput.

simulations and computed the averages of the values obtained. In particular, we measured the hit rates and throughput obtained by the different systems over the interval between days 60 and 70 with a non biased 5% cache and full cache sampling. The results are shown in Figure 9.

The figures show that SACS is able to outperform LRU and LFU regarding both hit rates and throughput. SACS tops LFU by, on average, 1%, and LRU by over 10%. Throughput follows a similar pattern, with SACS obtaining a small margin over LFU, and a larger margin over LRU. Overall, throughput values are over three times lower than the ones obtained with the sequential simulations, due to the overhead of concurrency control and network delays. These factors end up diluting the computational tradeoff between throughput and hit rate. In this case, throughput is proportional to hit rate, because the higher the hit rate, the more concurrent requests the cache is able to serve, whereas in the sequential execution of the simulations, throughput was dependent on the algorithmic overhead of the replacement strategy.

5 Related work

Due to the importance of the replacement algorithm for the caching system, a large body of work in the area of cache replacement can be found in literature. According to a survey by Podlipnig and Böszörmenyi [9], these algorithms can be grouped into five different categories. The first category consists of *recency based algorithms* [16,17,29], which use the time elapsed since the last request to a cached page as the main factor in their replacement decisions. LRU is the most representative of such algorithms and other algorithms are usually extensions of standard LRU. For example, *LRU-min* [16] tries to minimize the number of removed pages by applying LRU to a candidate list composed solely of pages that are larger than the most recently requested page. The main drawback of these algorithms is, as mentioned throughout the paper, the fact that they are oblivious to page popularity.

Frequency based algorithms [14,17] use request frequency as their main criteria. LFU is an example of this group of algorithms and serves as the model for other algorithms in the class. Because LFU is prone to cache pollution, several other algorithms employ aging techniques that aim at minimizing the frequency count of some or all of the pages, either periodically or at some specific event. However, it is questionable if aging techniques are better than recency based strategies [9]. SACS, on the other hand, by combining distance with recency and frequency, is able to handle cache pollution efficiently, without compromising the performance of the cache during standard (non polluted) operation.

Similarly to SACS, *recency-frequency based algorithms* [18,30,31] combine recency with frequency (and, possibly, other factors). SACS sets itself apart from the algorithms in this group by emphasizing distance over recency or frequency, which allows it to base its predictions of future accesses not only on past behaviour, but also on semantic knowledge.

Function based algorithms [15,32,33] use potentially general utility functions to evaluate the eviction potential of pages. Several such algorithms exist, with the common aspect that all use multiple weighted factors to score a page. Function based algorithms provide an elegant solution that has the potential to seamlessly adapt to

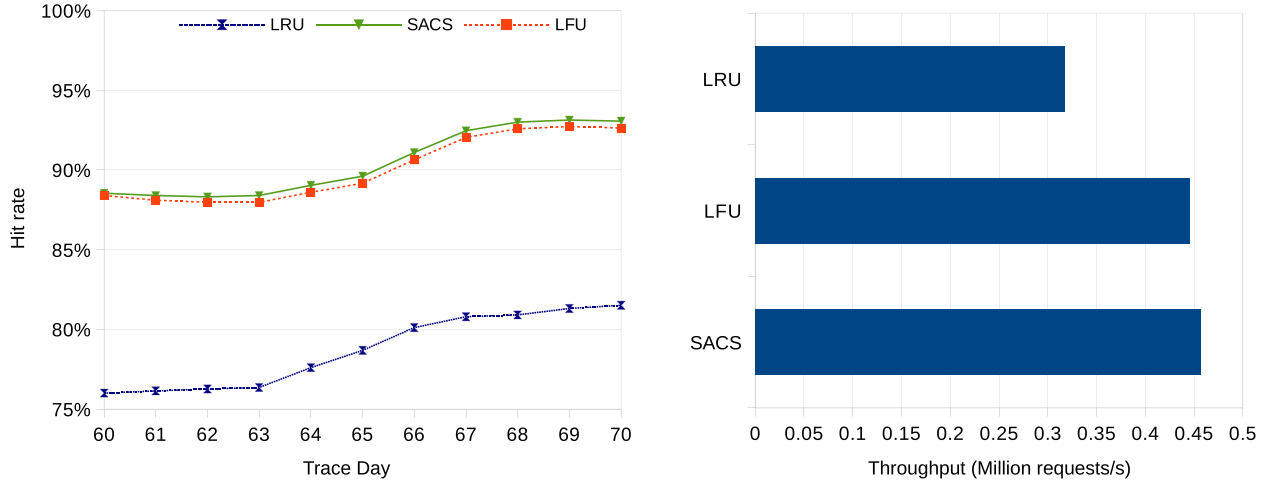

Figure 9 Hit rate and throughput for the network simulation scenario. (a) Hit rates. **(b)** Throughput.

different request patterns and workload dynamics. However, its main shortcoming lies in the difficulty of appropriately setting and tuning the parameters and weights of the utility function.

The final group of the taxonomy of Podlipnig and Böszörmenyi comprises algorithms which base their decisions on randomized choices [19,34,35], typically combined with other factors. *Randomized algorithms* were proposed to reduce the need for the complex data structures that are required by some of the traditional replacement algorithms. However, due to their intrinsic non-determinism, it is difficult to assert if their performance is consistent over time and under different request patterns and workloads.

In addition to the the groups identified by Podlipnig and Böszörmenyi, other authors have proposed replacement algorithms that employ machine learning techniques [11-13]. These algorithms include an offline learning phase in which the algorithm is trained with information from past requests (e.g., logs of a web server/cache). Then, the knowledge acquired in the learning phase is applied, at runtime, to choose the candidates for eviction. Unlike traditional cache replacement, machine learning algorithms are equipped with the mechanisms to predict future requests, rather than basing their decision only on past behaviour. However, their performance is tightly related with the data used during the training phase, which is itself solely based on past behaviour. SACS, on the other hand, looks into more immediate and contextual information which, as our evaluation results indicate, may prove useful in identifying future requests.

To the best of our knowledge, ours is the first work to look into the links inside web pages as a factor for replacement decisions in the context of web caching. By doing so, we base our predictions of future requests not only on past behaviour (by incorporating traditional techniques such as LRU and LFU), but also on semantic and contextual information. In addition, while in this paper we combine distance with standard LRU and LFU, our algorithm can easily be modified to have distance combined with other factors used in existing algorithms, including complex utility functions or randomized techniques.

Our solution is also related with the area of interest awareness in multiplayer games [36,37] and cooperative work [38,39]. Interest awareness systems manage the consistency of each user's view of the objects of a distributed application according to their distance to some special object(s). In multiplayer games, the consistency of game objects is stronger closer to each player's avatar and becomes weaker as the distance (within the virtual world) increases. In cooperative work applications, such as cooperative text editing and distributed software development, instead of metric distance, a *semantic distance* that measures the strength of the relations between different elements of a text document/software project is used. Similarly, our algorithm also assigns priorities according to distance, in an inversely proportional manner. However, in SACS, we do it to evaluate the eviction potential of web pages, rather than to determine the consistency requirements of distributed objects.

6 Conclusion

Caching of web pages is an optimization strategy that has been around since the early days of the web. Web caching improves server performance and response times by providing more than one source for each data item and/or placing contents closer to users. In this context, choosing which pages to keep in the cache and which to evict has a great impact on the performance of cache servers and, consequently, on the web as a whole. Traditional strategies used in practice base their decisions on static information and are, thus, vulnerable to the dynamism and unpredictability of user access patterns.

In this paper, we proposed SACS, a novel cache replacement algorithm that decides on which objects to remove based on the link navigation information of the cached pages. The design of SACS is inspired by the observation that the links contained in recently accessed pages are good indicators of future requests. SACS builds on this observation by assigning priorities to cached objects in such a way that objects whose distance to recently accessed pages (measured as the length of the shortest path of links between them) is high have higher probability of being removed, while pages closer to recently accessed pages are kept cached. Our solution additionally takes frequency into account, weighing it with distance when deciding on the eviction priority of each object. The evaluation results obtained and presented in this paper show that our solution is able to either match or even surpass the performance of existing algorithms in the scenarios in which they are better, while preventing their shortcomings in the scenario in which their performance falls short.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

AN designed and implemented the proposed system, conceived and carried out the evaluation and wrote the manuscript. CR implemented a preliminary version of the proposed system. LV and PF participated in the design of the system and revised the manuscript. All authors read and approved the final manuscript.

Acknowledgements

This work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2013.

Received: 30 September 2014 Accepted: 27 January 2015
Published online: 28 February 2015

References

1. Erman J, Gerber A, Hajiaghayi MT, Pei D, Spatscheck O (2009) Network-aware forward caching. In: Proceedings of the 18th International Conference on World Wide Web. WWW '09. ACM, New York, NY, USA. pp 291–300
2. Labovitz C, Iekel-Johnson S, McPherson D, Oberheide J, Jahanian F (2010) Internet inter-domain traffic. SIGCOMM Comput Commun Rev 40(4):75–86
3. Cardellini V, Casalicchio E, Colajanni M, Yu PS (2002) The state of the art in locally distributed web-server systems. ACM Comput Surv 34(2):263–311
4. Kanodia V, Knightly EW (2003) Ensuring latency targets in multiclass web servers. Parallel Distributed Syst IEEE Trans 14(1):84–93
5. Zhang Y, Ansari N, Wu M, Yu H (2012) On wide area network optimization. Commun Surv Tutor IEEE 14(4):1090–1113
6. Rabinovich M, Spatschek O (2002) Web Caching and Replication. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA
7. Davison BD (2001) A web caching primer. IEEE Internet Comput 5(4):38–45
8. Wang J (1999) A survey of web caching schemes for the internet. ACM SIGCOMM Comput Commun Rev 29(5):36–46
9. Podlipnig S, Böszörmenyi L (2003) A survey of web cache replacement strategies. ACM Comput Surv (CSUR) 35(4):374–398
10. Romano S, ElAarag H (2008) A quantitative study of recency and frequency based web cache replacement strategies. In: Proceedings of the 11th Communications and Networking Simulation Symposium. CNS '08. ACM, New York, NY, USA. pp 70–78
11. Ali W, Shamsuddin SM, Ismail AS (2012) Intelligent web proxy caching approaches based on machine learning techniques. Decis Support Syst 53(3):565–579
12. Cobb J, ElAarag H (2008) Web proxy cache replacement scheme based on back-propagation neural network. J Syst Softw 81(9):1539–1558
13. Koskela T, Heikkonen J, Kaski K (2003) Web cache optimization with nonlinear model using object features. Comput Netw 43(6):805–817
14. Arlitt M, Cherkasova L, Dilley J, Friedrich R, Jin T (2000) Evaluating content management techniques for web proxy caches. ACM SIGMETRICS Perform Eval Rev 27(4):3–11
15. Cao P, Irani S (1997) Cost-aware WWW proxy caching algorithms. In: 1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997. USENIX Association, Berkley, California, USA
16. Abrams M, Standridge C. R, Abdulla G, Williams S, Fox EA (1995) Caching proxies: Limitations and potentials. In: Proceedings of the 4th International Conference on World Wide Web. WWW '95, Boston, Massachusetts, USA
17. Zhang J, Izmailov R, Reininger D, Ott M, A NUS (1999) Web caching framework: Analytical models and beyond. In: Proceedings of the 1999 IEEE Workshop on Internet Applications. WIAPP '99. IEEE Computer Society, Washington, DC, USA. p 132
18. Cheng K, Kambayashi Y (2000) Lru-sp: A size-adjusted and popularity-aware lru replacement algorithm for web caching. In: 24th International Computer Software and Applications Conference. COMPSAC '00. IEEE Computer Society, Washington, DC, USA. pp 48–53
19. Starobinski D, Tse D (2001) Probabilistic methods for web caching. Perform Eval 46(2-3):125–137
20. Bahn H, Koh K, Noh SH, Lyul SM (2002) Efficient replacement of nonuniform objects in web caches. Computer 35(6):65–73
21. Scheuermann P, Shim J, Vingralek R (1997) A case for delay-conscious caching of web documents. In: Proceedings of the Sixth International World Wide Web Conference, Santa Clara, California USA
22. Ehcache - Performance at any scale. http://ehcache.org/
23. HTML Parser. http://htmlparser.sourceforge.net/
24. Netty project. http://netty.io/
25. Arlitt M, Jin T 1998 World Cup Web Site Access Logs. http://www.acm.org/sigcomm/ITA/
26. Internet Archive. https://archive.org/
27. Wong K-Y (2006) Web cache replacement policies: a pragmatic approach. IEEE Network 20(1):28–34
28. Squid: Optimising Web Delivery. www.squid-cache.org
29. Aggarwal C, Wolf JL, Yu PS (1999) Caching on the world wide web. IEEE Trans Knowl Data Eng 11(1):94–107
30. Menaud J-M, Issarny V, Banâtre M (1999) Improving the effectiveness of web caching. In: Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems. Springer, London, UK, UK. pp 375–401
31. Osawa N, Yuba T, Hakozaki K (1997) Generational replacement schemes for a www caching proxy server. In: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking. HPCN Europe '97. Springer, London, UK, UK. pp 940–949
32. Cohen E, Krishnamurthy B, Rexford J (1998) Evaluating server-assisted cache replacement in the web. In: Proceedings of the 6th Annual European Symposium on Algorithms. ESA '98. Springer, London, UK, UK. pp 307–319
33. Jin S, Bestavros A (2001) Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. Comput Commun 24(2):174–183
34. Hosseini-Khayat S (1998) Replacement algorithms for object caching. In: Proceedings of the 1998 ACM Symposium on Applied Computing. SAC '98. ACM, New York, NY, USA. pp 90–97
35. Psounis K, Prabhakar B (2002) Efficient randomized web-cache replacement schemes using samples from past eviction times. IEEE/ACM Trans Netw 10(4):441–455
36. Bezerra CE, Cecin FR, Geyer CFR (2008) A3: A novel interest management algorithm for distributed simulations of mmogs. In: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications. DS-RT '08. IEEE Computer Society, Washington, DC, USA. pp 35–42
37. Veiga L, Negrão A, Santos N, Ferreira P (2010) Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. J Int Serv Appl 1(2):95–115
38. Negrão AP, Costa J, Ferreira P, Veiga L (2014) Interest aware consistency for cooperative editing in heterogeneous environments. Int J Cooperative Inform Syst 23(1):42–75
39. Negrão A, Mateus M, Ferreira P, Veiga L (2013) Adaptive consistency and awareness support for distributed software development. In: 21st International Conference on Cooperative Information Systems (CoopIS 2013) – On the Move to Meaningful Internet Systems: OTM 2013 Conferences. Lecture Notes in Computer Science, Graz, Austria Vol. 8185. pp 259–266

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com