The Least Semantically Related Cache Replacement Algorithm*

Alcides Calsavara
Rogério Guaraci dos Santos
Edgard Jamhour
Pontifícia Universidade Católica do Paraná
Programa de Pós-Graduação em Informática Aplicada
Rua Imaculada Conceição, 1155, Prado Velho, Curitiba, PR, Brazil
{alcides,rogerio,jamhour}@ppgia.pucpr.br

Abstract

This article presents the Web cache replacement algorithm named Least Semantically Related (LSR), as an alternative to well-known and widely employed replacement policies, such as SIZE, LFU and LRU, which are based on physical properties of objects (documents). LSR is based on the semantics of the information contained within objects: LSR tends to evict objects which are less related to the last object to enter the cache with respect to their semantics, therefore preserving objects of more interest to clients. A detailed algorithm and a data structure for LSR where objects are classified according to a previously established taxonomy were designed and implemented for the purposes of validation and comparison with other replacement policies. Besides, a framework for simulation was designed and verified, including the data preparation process. The initial experimental results show that LSR outperforms traditional policies in terms of hit rate, in most cases.

LANC'03 October 3-5, 2003, La Paz, Bolivia.
 ISBN 1-58113-789-3/03/0010
Order number: 103035. Fee: $5.00.

1 Introduction

This article introduces a novel cache replacement algorithm based on document semantics, i.e., some information about the contents of the documents. This new algorithm is called *Least Semantically Related* – LSR, for short. The main supposition of LSR is that every client tends, for a certain period of time, to seek documents which are related to a given subject and, therefore, have close semantics. That is accomplished by simply discarding objects which are less related to a new entry with respect to their semantics: LSR favors the permanence of documents in cache with are close with respect to their semantics and discard documents which might be of less interest to a client. In this first version, objects are classified according to a previously established taxonomy.

The increasing use of the Internet and its emerging applications make it to tend to saturation due to the corresponding increase in data transmission. Cache mechanisms help to reduce network and server load by avoiding transmission of documents which are requested several times. They are traditionally employed on three places: *(i)* between a client (e.g., a Web browser) and a server; *(ii)* between a client and a proxy ; and *(iii)* between proxies. In fact, cache mechanisms are currently fundamental for the scalability of the Internet.

However, every cache is naturally limited in size and, as a consequence, documents may

need to be discarded to create room for a new entry. That is, it may be necessary to replace some documents by the new one. For that purpose, there is a number of cache replacement algorithms currently in use [Arlitt et al., 1998]. Typically, these algorithms are based on physical properties of documents, such as their size, their access frequency rate, their last time of access, and so on, as discussed in Section 2. In other words, the traditional cache mechanisms treat documents as black boxes and discard them according to some information that is not related to their contents at all. Nevertheless they produce satisfactory results in average, a question that remains open is whether an algorithm based on the contents of documents would be more efficient.

A summary of our research work is here presented; a more detailed discussion is presented in [dos Santos, 2001]. The remaing of this article is organised as follows. Section 2 discusses the main cache replacement algorithms currently in use on the Internet. Section 3 describes LSR – the proposed algorithm – and presents an example of use. Section 4 describes an implementation of LSR where document semantics is realised through tree-based classification of documents. Section 5 describes an experiment which simulates LSR for the purposes of validation and comparison to traditional cache replacement algorithms. Finally, Section 6 presents some conclusions and gives some directions for future work.

2 Related Work

Cache replacement algorithms have a fundamental role in the project of any storage component. Such algorithms, for example, have been intensively studied in the context of virtual memory management systems [O'Neil et al., 1993]. There are dozens of such algorithms documented in the literature. Our intention in this Section is just emphasize that they all use physical properties of objects as the criteria to evict objects. We neither analyze nor compare such algorithms since their own bibliographical refecences do that.

We do not describe any Web cache replacement algorithm that is based on document semantics because, to the best of our knowledge, none is well known. There are, however, some attempts to create semantic caching mechanisms for database and query systems, where data is well-structured, thus providing a means of organizing objects according to some semantic information. Some of those attempts include [Dar et al., 1996], [Chidlovskii et al., 1999], [Keller and Basu, 1996] and [Ren et al., 2003]. They all conclude that the semantic-based approach to cache replacement performs better than traditional policies.

More recently, in the field of wireless infrastructure and applications, there is a preocupation to develop more appropriated caching schemes where physical locality can be exploited. The research work presented in [Zheng et al., 2002] combines such an idea with information semantics and obtains good results as well.

Finally, there are efforts to create caching mechanisms for distributed objects systems, such as [Atzmon et al., 2002], where an object can encapsulate a Web document; it builds a hierarcy for each cached object based on client's access pattern and objects are explicty evited by clients by invoking operations to reregister objects.

For completeness sake, we briefly describe the most relevant cache replacement algorithms which are based on physical properties of objects, because either they are widely employed on the Internet or their research stage is quite mature.

SIZE This algorithm simply elects for removal the biggest object in cache. When two objects with the same size are elected, the less frequently accessed is removed [Aggarwal et al., 1999].

LEAST RECENTLY USED (LRU) This algorithm elects for removal the object which is the one least recently used by clients [Aggarwal et al., 1999].

LEAST FREQUENTLY USED (LFU) This algorithm elects for removal the object which is the one least frequently used by clients [Williams et al., 1996]. There are two versions of this algorithm:

1. *In-Cache LFU*: In this version, the

access counter for an object is zeroed everytime the object enters the cache.

2. *Perfec LFU*: In this version, the access counter for an object is zeroed only the first time the object enters the cache. If an object that has been previously removed comes back to cache, its access counter will have the same value when removed.

LOWEST RELATIVE VALUE (LRV) This algorithm assigns a relative cost value for each object in cache in order to calculate their utility; the object with the lowest relative value is elected for removal [Rizzo and Vicisano, 2000].

LRUMIN This algorithm tends to keep in cache objects of smaller size in order to minimize the number of replacements. Suppose that a new object of size S has to enter the cache and there is not enough room for it. If there is any objects in cache which size is at least S, the least recently used (LRU) one is removed. Otherwise, objects with size at least $S/2$ are sequentially removed following the LRU policy. If still necessary, objects with size at least $S/4$, $S/8$, and so on are removed until the needed room is created [Aggarwal et al., 1999].

GREEDYDUAL-SIZE (GD-SIZE) This algorithm is a generalization of the LRU. It is concerned with the case when objects have the same size, but incur different costs to fetch from a secondary storage. The algorithm associates a value, H, with each cached object. The initial value of H is the cost to fetch the object. When a replacement need to be made, the object with the lowest H, say h, is replaced, and then all objects reduce their H values by h. Everytime an object is accessed, its H value is restored to the corresponding original value [Cao and Irani, 1997].

HYPER-G This algorithm is a refinement of the LFU, with last access time and size considerations [Williams et al., 1996].

PITKOW/RECKER This algorithm removes the least-recently-used document, except if all documents are accessed today, in which case the largest one is removed [Williams et al., 1996].

LOWEST-LATENCY-FIRST This algorithm tries to minimize average latency by removing the object with the lowest download latency first [Wooster and Abrams, 1997].

HYBRID This algorithm aims at reducing the total latency [Cao and Irani, 1997]. It replaces the object which results the lowest value for the following function:

$$\frac{(c_s + \frac{W_b}{b_s}) \times (n_p)^{W_n}}{z_p}$$

where s is a server, p is an object located in s, c_s is the time to connect to s, b_s the bandwidth to server s, n_p is the number of times p has been requested since it was brought into the cache, z_p is the size (in bytes) of p, and W_b and W_n are constants.

FIRST-IN, FIRST-OUT (FIFO) This algorithm replaces the object that enters the cache first [Silberschatz and Galvin, 1994].

FUNCTION-BASED REPLACEMENT This algorithm employes a general function for different factors, such as the last access, entry time of an object into cache, transfer cost and the time-to-live of an object [Wooster and Abrams, 1997].

SIZE ADJUSTMENT LRU (SLRU) This algorithm is known as the *Knapsack Problem* – it orders the objects in cache according to their cost and size; the object with largest index is evicted from cache when a replacement is needed [Aggarwal et al., 1999].

Pyramidal Selection Scheme with award This policy classifies objects according to their size using a logarithmic function, combined with frequency access rate. [Cheng and Kambayashi, 2000].

Partitioned Cache This policy splits the cache into partitions that sotre classes of documents based on their size, instead of

having a single cache to store all documents. Its aim is to take into consideration the high variability noticed in the WWW [Murta et al., 1998].

As discussed above, the known algorithms for cache replacement are based only on physical properties of objects, and each algorithm uses a particular heuristic to elect objects to remove from cache. Therefore, there is a condition where such an algorithm performs optimally, as well as there is a condition where the same algorithm performs badly (worst case), that is, there is no optimal solution for the problem and the choice of the most appropriate algorithm will depend always on the user access pattern.

3 The LSR algorithm

3.1 Rationale

The Least-Semantically Related (LSR) algorithm associates a semantics to each object in cache in such a way that is possible to determine a semantics distance between any two objects; the semantics of an object is (staticaly or dynamicaly) defined according to its contents. When a new object n enters the cache, the algorithm evicts the objects from cache which are less semantically related to n, that is, the objects which are more distant to n with respect to their semantics.

3.2 Formalization

The LSR algorithm to insert a new object into cache is formalized through the pseudo-code listed in Figure 1. The algorithm has two input arguments (line 1): a cache C where an object n must be inserted. The test in line 2 checks whether the object fits in the cache; if the object size is greater than the cache capacity then there is nothing to be done. The code between lines 3 and 8 takes care of object removal, if necessary, while line 9 contains the code to insert the new object into the cache. The test in line 3 checks whether is necessary to do any object removal; while the object size is greater than the free room in the cache, object removal will be carried out. The set of objects D defined in line 4 is a subset of the cache C that contains the most semantically distant objects with respect to the new object n still remaining in the cache; all objects in D have exactly the same semantics distance to n. The loop between lines 5 and 8 does the actual work: it evicts objects from cache according to the subset D, until there is no objects left in D or the free room in cache is enough to insert the new object. The code in line 6 picks an object from the subset D according to some implementation-dependent criteria, since they all have the same semantics distance to the new object n. Thus, any of the traditional algorithms, such as the LRU, could be applied in this case.

4 Tree-based implementation

One approach to associate semantics to each object is through a taxonomy, i.e., objects can be organized according to a tree-based hierarchy of subjects: each tree node represents a subject and contains objects whose contents have the corresponding semantics, as well as a tree node may have children, which represent more specific subjects.

The LSR algorithm described in Section 3 must be refined according to approach used to represent object semantics. In the case where a tree-based hierarchy of subjects is used to represent semantics, it is assumed that there is a function that, when applied to an object, returns the semantics of such an object in the form of a sequence of tree nodes that represent subjects, from more general to more specific ones. Naturally, this is not a rule for the Internet as a whole in the present days, but there are efforts to create a standard taxonomy for Internet objects, such as the Open Directory Project[1]. Also, there is a proposal [Shmidt, 2002] based on the RDF (Resource Description Format) standard to create Web servers that provide the semantics of an object given its URL. Moreover, it is perfectly feasible to create a taxonomy for more controlled environments, such as for a digital library, for a specific domain of knowledge or for a certain organization, where LSR could then be

[1] URL: http://www.dmoz.org

```
1 lsr ( Cache C, Object n )
2    if n.size <= C.capacity then
3        while n.size > C.freeroom
4            Set D := C.get_most_semantically_distant_objects( n );
5            repeat
6                Object x := D.remove_an_object;
7                C.evict_object( x );
8            until D is empty or C.freeroom >= n.size
9        C.insert( n );
```

Figure 1: Pseudo-code for the LSR algorithm

employed.

4.1 Example

The diagram in Figure 2 illustrates a tree-based hierarchy of subjects and corresponding objects in cache. The node named *Root* represents the universe of subjects and has two children named *Sport* and *Music*. The node *Sport* has two children named *Basketball* and *Football*. The node *Music* has two children named *Jazz* and *Rock*. Thus, each object in cache may be linked to its suject by inserting the object under the corresponding tree node. For example, object number 1 is linked to subject *Root.Sport*, while object number 6 is linked to subject *Root.Music.Rock*. A node named *etc* is present in each tree branch in order to cover all subjects which are absent. Thus, for example, if an object whose contents has the semantics *Root.Sport.Football.EuroCup*, like the case of the object number 5, such an object should be placed under the node *Root.Sport.Football.etc*, since the subject *EuroCup* is absent.

4.2 Algorithm refinement

The only part of the LSR algorithm described in Section 3 that needs refinement is the code present in line 4 of Figure 1, i.e., the function *get_most_semantically_distant_objects*, which returns the set of objects in cache that are the most semantically distant from the object given as argument. For the tree-based hierarchy of subjects approach to semantics representation, such a refinement is formalized through the pseudo-code listed in Figure 3.

The function *get_most_semantically_distant_objects* has two arguments as input (line 1): a cache C where an object n should be inserted. Note that the object is not inserted by this function; it is given as argument for the function to know where in the tree of subjects the object should be inserted, and then determine the objects that are most semantically distant from such a place; the set of objects (an instance of *ObjectSet*) obtained as most distant is returned by the function. The set of objects named *result* defined on line 2 is used for collecting all objects in cache that have the maximum semantics distance from the object n, and is returned on line 6. The function *mark_ancestors* invoked on line 3 gets the semantics of the object given as argument and marks as "ancestors" all nodes in the tree corresponding to the list of subjects contained by that semantics, including the node where the object should be inserted. The sequence of marked nodes defines the **ancestors path** for the new object; it denotes the set of objects that have close semantics to the new object and, therefore, should be the last to be evicted from cache. The function *unmark_ancestors* invoked on line 5 removes the marks done before. The recursive function *collect* (detailed below) invoked on line 4 inserts in *result* all objects that have the maximum semantics distance from the new object, starting from the *Root* subject, and considering the ancestors path.

The function *collect* has two arguments as input (line 7): a *subject* (or node) where the function is applied and a set of objects *result* where the most semantically distant objects

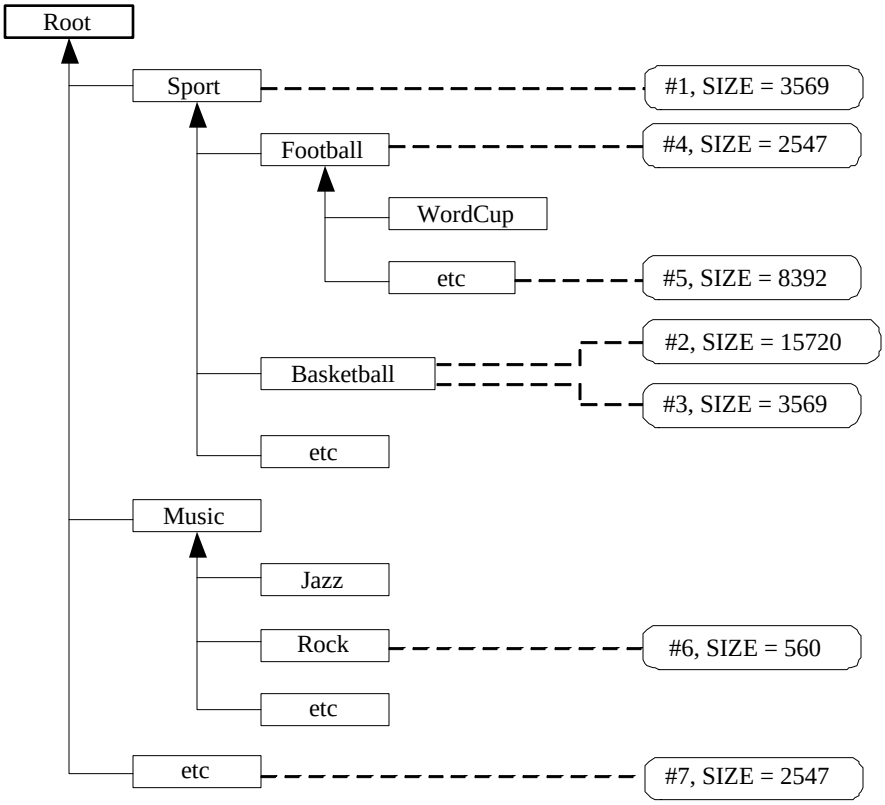

Figure 2: A tree-based hierarchy of subjects and objects in cache

must be inserted. The "deepness" of a node is given by the number of tree levels of descendants until the lowest descendant that contains at least one object; the deepness of the lowest descendant is 1, the deepness of its father is 2, and so on. The algorithm gives priority to collect objects contained by deepper subjects, as an attempt to evict objects that have the most specific semantics, therefore, most distant from the new object. The target *subject* may have no childreen (directly or indirectly) containing objects (the *deepness* of such a child would be greater than zero). If this is the case – as verified by the code on line 8 – objects contained by the target *subject* itself is added to the *result* set, as shown by the code on line 9. Otherwise, the children's objects should be collected as follows. The set of subjects *candidates* defined on line 11 is initialized with the "deepper" children of *subject*, excluding subjects marked as ancestors. The function *get_deepper_but_ancestors* invoked on line 11 ignores not only ancestor subjects but also subjects whose deepness is 0. The obtained set of candidates will be empty when the only child is marked as ancestor. Such a condition is verified by the code on line 12, while the code on line 13 assigns as the only candidate the child that is marked as ancestor. Once the set of candidates is defined, the function *collect* is recursively invoked for each one.

4.3 Example of application of LSR for a tree of subjects

Figure 4 shows a tree of subjects with corresponding objects before and after the insertion of a new object into the cache. The new object is identified as #1, its semantics is $G.H.A$ and its size is 19. The capacity of the cache is 30, while its size before insertion is 29. That is, the required room is 19, while the free room is 1. Thus, some objects have to be evicted from cache to enable the insertion. More precisely, it is necessary to evict objects to create extra room of size at least 18. In this example, the function *remove_an_object* (Figure 1, line 6) is implemented using the SIZE policy. Table 1 shows step-by-step how data concerning LSR changes until the object is finally inserted into cache.

```
01 get_most_semantically_distant_objects( Cache C, Object n ) : ObjectSet
02     ObjectSet result := empty;
03     mark_ancestors( n );
04     collect( C.Root, result );
05     unmark_ancestors;
06     return result;
```

```
07 collect( Subject subject, ObjectSet result )
08     if subject has no child with deepness > 0
09         result.add( subject.objects );
10     else
11         SubjectSet candidates := subject.children.get_deepper_but_ancestors;
12         if candidates is empty
13             candidates := subject.children.get_ancestor;
14         for each s in candidates
15             do collect( s, result );
```

Figure 3: Pseudo-code for the LSR refinement: function to obtain most semantically distant objects on a tree of subjects

5 Experiment and validation

The LSR Algorithm was implemented for validation and performance assessment purposes. This Section presents the data preparation process for simulation, such as some results. The algorithm is compared to traditional object replacement policies such as LFU, LRU and SIZE, with respect to hit rate, since they are widely employed on the Internet and there are many studies that compare them, such as in [Arlitt et al., 1998]. A module named *Internet Client* represents any Internet entity that uses a cache, such as a Web browser, a proxy or even an Internet server. Since ordinary Internet objects do not provide explicit semantics, the simulated sequence of accesses contains only objects defined by the Yahoo! search engine, where objects are properly classified according to a certain taxonomy. The *Internet Client* module simulates a sequence of object accesses according to an expected user access pattern, i.e., it is supposed that users access objects of a certain subject for some time before changing to another subject – *locality*. Moreover, the simulated object accesses conform to a real distribution of objects per number of accesses, obtained from a Squid Proxy[2] log file; the Monte Carlo [Liu, 2001] Method is employed to implement such a distribution by categorizing Yahoo! objects according to the number of times they should be accessed in a certain period of time. The only relevant object attributes for the purposes of simulation are **semantics** (given by the Yahoo! taxonomy) and **size** (obtainable, for example, by actually accessing an object through its URL). The second module, named *Cache Manager*, contains the cache where objects are stored and a corresponding tree of subjects; the LSR Algorithm is implemented by this module. The simulation process is fully described by the following sections.

5.1 Universe of objects

The first step for simulating the LSR Algorithm is to define a *universe of objects*, which corresponds to the set of objects available for access. In this very first experiment with LSR,

[2] www.squid-cache.org

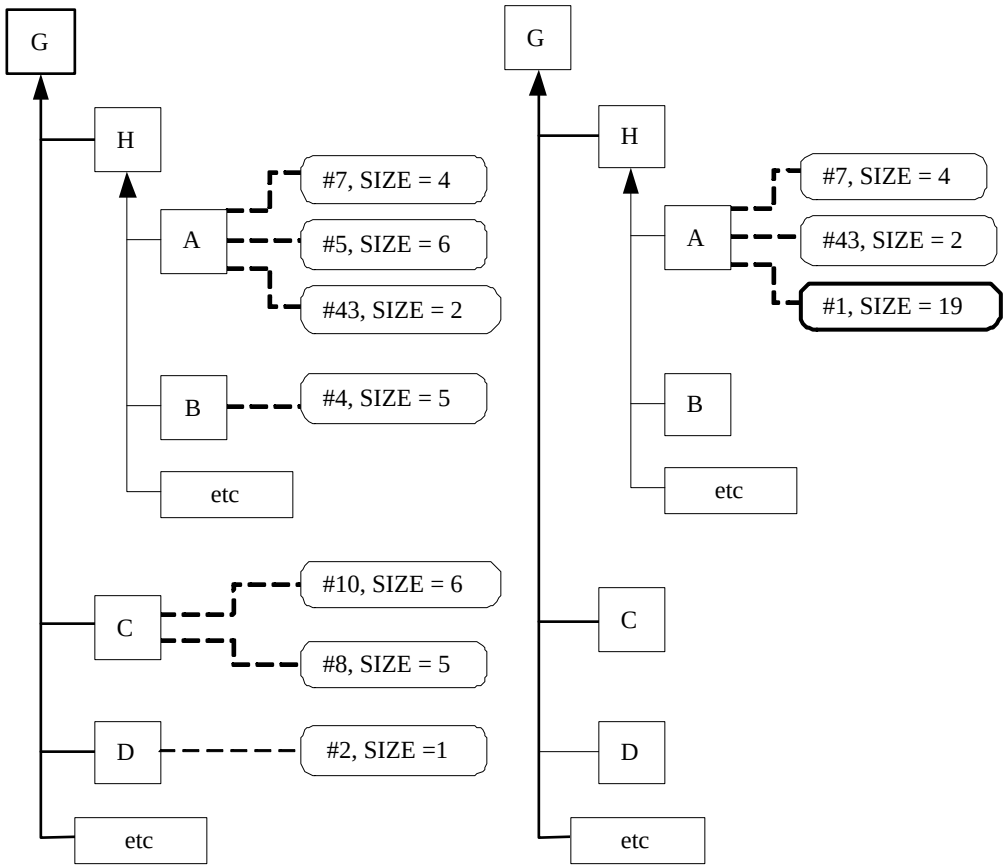

Figure 4: A tree of subjects before and after a new object enters the cache

such a universe of objects was manually built by reproducing part of the Yahoo! search engine hierarchy of subjects. More precisely, a universe of 983 objects and 180 distinct subject paths was built. The following data are necessary for each object for the purposes of simulation: its semantics (subject path), its identity (normally, its URL) and its size (normally, in bytes). Thus, two plain text files were created as input: a file containg the universe of objects for an instance of the Internet Client module, and a file containing all corresponding distinct subject paths for an instance of the Cache Manager module.

5.2 Distribution of object accesses

The Internet Client module produces a sequence of object accesses. Such a sequence must conform to real-world object access patterns. A typical distribution of object accesses was discovered by analysing the log file produced by a Squid Proxy for a certain period of time, containing about 2 million records of access to 866,915 Internet objects. Figure 5 shows a histogram with the corresponding distribution. It can be noticed that many objects are accessed a few times, while a few objects are accessed many times. This actual distribution was the basis to generate a sequence of object accesses for the universe of objects chosen. This was implemented by associating each object in the universe with a *category of access*, according to the actual distribution of accesses.

5.3 Sequence of object accesses

The next step for simulating the LSR Algorithm is to create a bounded sequence of object accesses corresponding to a possible behaviour of an Internet client. Every item of the sequence fully describes an object: semantics, identity and size. Figure 6 shows a didactid example of sequence of accesses to objects classified according to the tree of subjects shown in

		while - step 1				*while - step 2*		*while - step 3*		*insert*
			repeat				*repeat*		*repeat*	
C.size	29		23	18	17		12		6	25
C.freeroom	1		7	12	13		18		24	5
D		{#10, #8, #2}	{#8, #2}	{#2}	{}	{#4}	{}	{#7, #5, #43}	{#7, #43}	
x			#10	#8	#2		#4		#5	
x.size			6	5	1		5		6	

Table 1: Example of step-by-step changes of LSR data

Figure 2. Figure 7 shows an excerpt of the sequence of object accesses actually used for the simulation. In both cases, it can noticed that some objects are accessed more than once, like in a real-world sequence of accesses. This is achieved by applying the Monte Carlo Method. Table 2 contains a small example of the necessary data to apply the Monte Carlo Method to obtain a sequence of accesses. Each row corresponds to a category of access (c) and contains the following data:

- the corresponding quantity (q) of objects in the universe of objects
- a probability obtained by dividing the quantity of objects by the total number of objects in the universe:

$$P_c = \frac{q_c}{\sum_c q_c}$$

- a lower bound obtained by summing the probabilities of all previous categories:

$$\sum_{i=1}^{c-1} P_i$$

- an upper bound obtained by summing the probabilities of all previous categories and the probability of this category too:

$$\sum_{i=1}^{c} P_i$$

Once such a table is built, objects are randomly chosen from the universe of objects to create a sequence of object accesses. There is a counter associated to each object: it grows by one every time the object is chosen; when it reaches the value of the corresponding category of access, the object is marked as *invalid* so it cannot be chosen again.

5.4 Results

A standard metrics for the efficacy of a cache system is the so-called *hit rate*, which corresponds to the probability of finding in cache a certain object requested by an Internet client. The hit rate of a cache system depends on the cache size and on the present number of accesses. For that reason, a number of sequences of object accesses was produced and submitted for simulation in caches of different sizes. More precisely, the experiment was carried out for thirteen sequences of accesses (varying in size from 1,000 to 30,000 accesses), six different sizes for cache (from 1 million to 10 million bytes) and four policies (LSR, SIZE, LRU and LFU), giving a total of 312 simulation scenarios. Each one of the thirteen sequences was rearranged in order to reflect the assumed *locality* property, that is, users access objects of a certain subject for some time before changing to another subject.

The simulation confirmed that all the policies tend to make the best use of the cache, i.e., the cache stays close to full after a number of object accesses; that shows that our implementation of the policies are correct in that sense. The graphics in Figure 8 shows the effect of number of object accesses on the hit rate for a cache of a certain size: it grows, tending to stabilize, as the number of object accesses grows. The graphics in Figure 9 shows the effect of the cache size on the hit rate: it grows as the cache size grows and, after a certain cache size, the hit rate is identical for all the policies, as expected – em theory, the cache can be big enough to store all objects of the universe.

The most important fact observed is that, practically in all simulation scenarios, the LSR

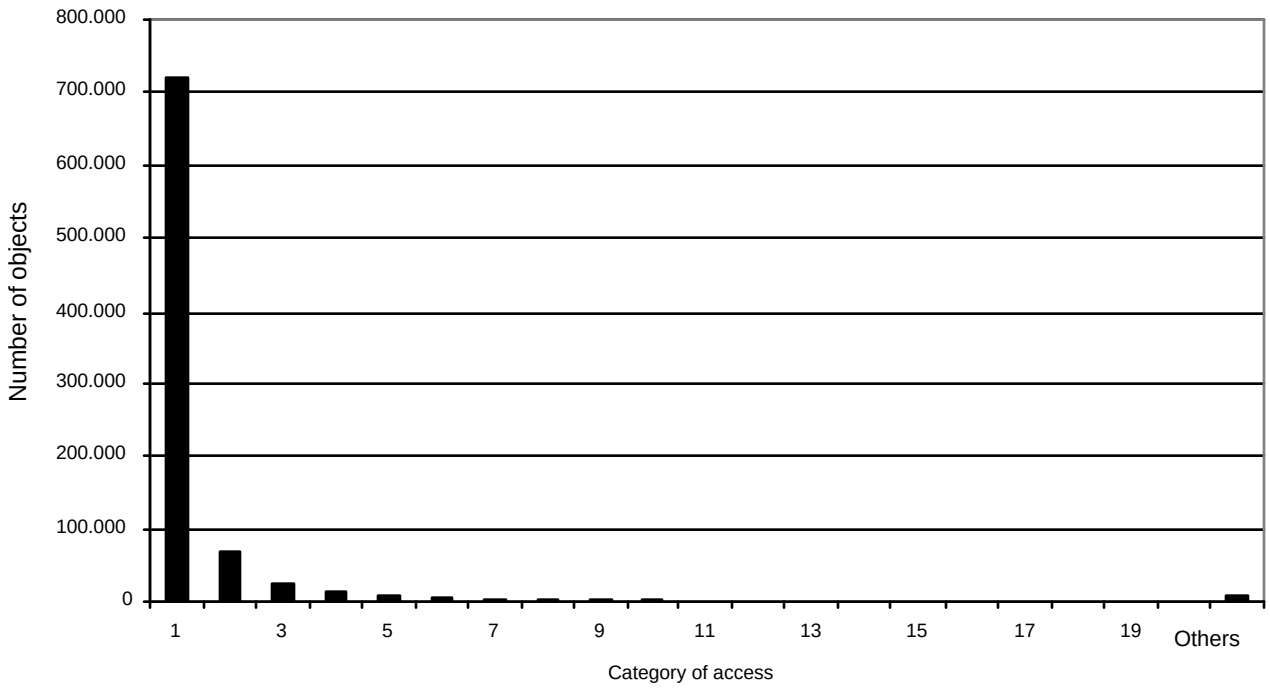

Figure 5: Distribution of object accesses according to a Squid Proxy log file

Algorithm perfomed – in terms of hit rate – better than the other policies. Although the actual gain by using LSR still needs new experiments that use larger data for an accurate assess, it is sure that such a gain exists.

6 Concluding remarks and future work

We have introduced a novel policy for object cache replacement, the LSR Algorithm, and shown that it performs better than traditional policies normally employed on the Internet, namelly SIZE, LFU and LRU, in terms of hit rate and, certainly, implies a gain in terms of network load. The algorithm was properly formalized and implemented for the purposes of validation and performance analysis. We have designed and implemented a simulation model that uses real-world data; since the algorithm assumes that it is possible to get the semantics for every Internet object, our first implementation is based on a tree of subjects to categorize objects and is based on the taxonomy given by the Yahoo! search engine.

The first results are very promissing, thus encouraging to proceed with the research. We intend to carry many improvements, such as:

- The development of LSR assumes that the semantics of each object is available. This should be feasible with the adoption of the RDF standard. A proposal by [Shmidt, 2002] explores this idea and shows that it is possible to use LSR for real-world systems. Certainly, that will require some implementation effort, but it will make it possible to test and assess LSR with a large amount of objects and users.

- The experiments done so far test an isolated cache. A possible extension is to experiment with a hierachy (or grid) of cooperating caches. For that purpose, we are considering to use the simulation platform developed by [Brandão and Anido, 2001].

- The tree of subjects approach used in the first implementation needs to be arranged

```
Root.Sport;#1;3569
Root.Sport.Football;#4;2547
Root.Music.Rock;#6;560
Root.Sport.Basketball;#2;15720
Root.Sport.Basketball;#3;3569
Root.Computer.ProgrammingLanguages;#7;2547
Root.Music.Rock;#6;560
Root.Sport.Football;#4;2547
```

Figure 6: A sequence of object accesses

```
Root.Arts.Artists.Collage and Assemblage;http://www.rowenadugdale.com/;5482
Root.Arts.Art History.Organizations;http://www.indiana.edu/~aah/;814
Root.Arts.Artists.Ceramics;http://ninacambron.com/;1207
Root.Computers & Internet.Desktop Publishing;http://www.dtp.com/;64140
Root.Computers & Internet.Desktop Customization;http://www.3dtop.com/;6139
Root.Computers & Internet.Desktop Publishing;http://www.dtpjournal.com/;11806
Root.Computers & Internet.Desktop Publishing;http://www.dtp.com/;64140
Root.Education.Equity;http://www.csd.uwa.edu.au/HERDSA/abstract/equity.html;1493
Root.Education.Early Childhood Education;http://www.ecewebguide.com/;13320
Root.Education.Bilingual;http://gopher.ael.org/~eric/digests/edorc968.html;15135
Root.Education.Chats and Forums;http://www.heproc.org/;14084
Root.Arts.Artists.Ceramics;http://ninacambron.com/;1207
Root.Entertainment.Cool Links.Business and Economy;http://www.adflip.com/;28675
Root.Entertainment.Cool Links.Business and Economy;http://www.barcar.com/;4478
Root.Computers & Internet.Desktop Publishing;http://www.dtp.com/;64140
Root.Social Science.Disability Studies;http://www.uic.edu/orgs/sds/;10300
Root.Social Science.Economics;http://www.sims.berkeley.edu/resources/infoecon/;4242
Root.Social Science.Area Studies;http://humanities.uchicago.edu/cis/;2249
```

Figure 7: A sequence of Yahoo! object accesses

according to users interests; LSR will perform better when the tree is well suited for users interests. Since such interests can change over time, it is required a rearrangement of the tree. Therefore, it would be interesting to have a mechanism that dynamically updates trees of subjects according to the way users change their interests. This is possible to dected by tracing the semantics of objects accessed. For example, if a *etc* node becames to busy, it means that the tree does not conform to users interests anymore.

- The current version of LSR is well suited for a single interest at a time (not necessarily a single user). An interesting research path is to enhance LSR to suit a number of parallel threads of interest (of a single user or a group of users). That will require extra information, such as history of access, that is, when objects need to be evicted from a cache, LSR would consider not only the semantics of the new object that enters the cache, but a window of latest objects. The research work presented in [Vakali, 2001] has already shown the benefits of keeping history information about object access when applied to traditional cache replacement policies.

- Another research path is to employ an alternative to tree of subjects as a means to represent semantics. We are currently

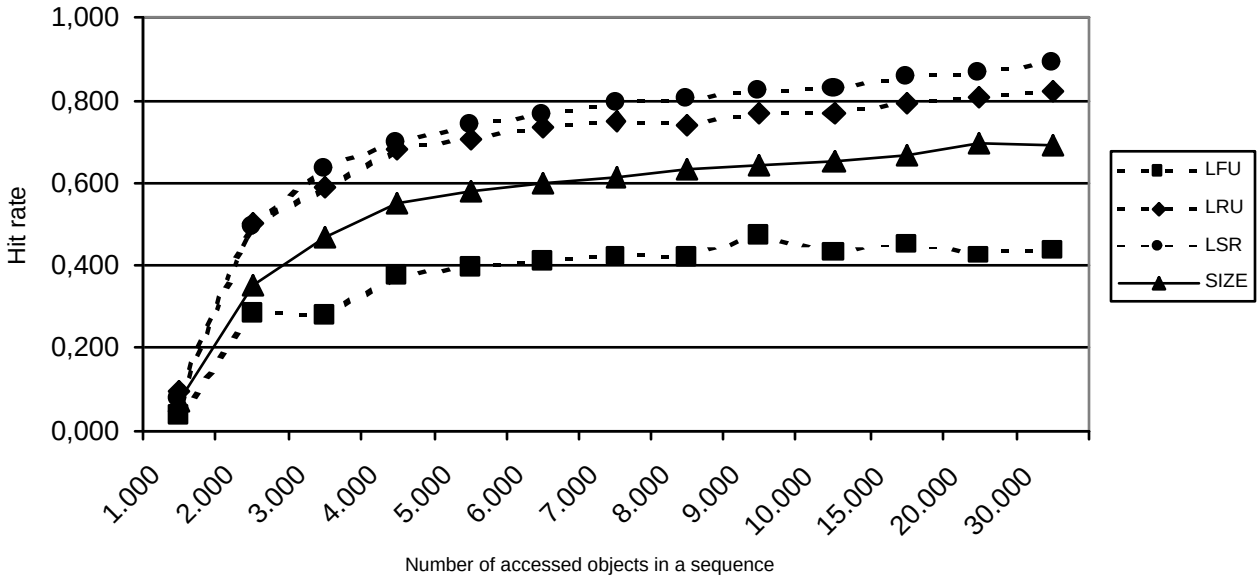

Figure 8: Hit rate for a cache of size 2,000,000 bytes

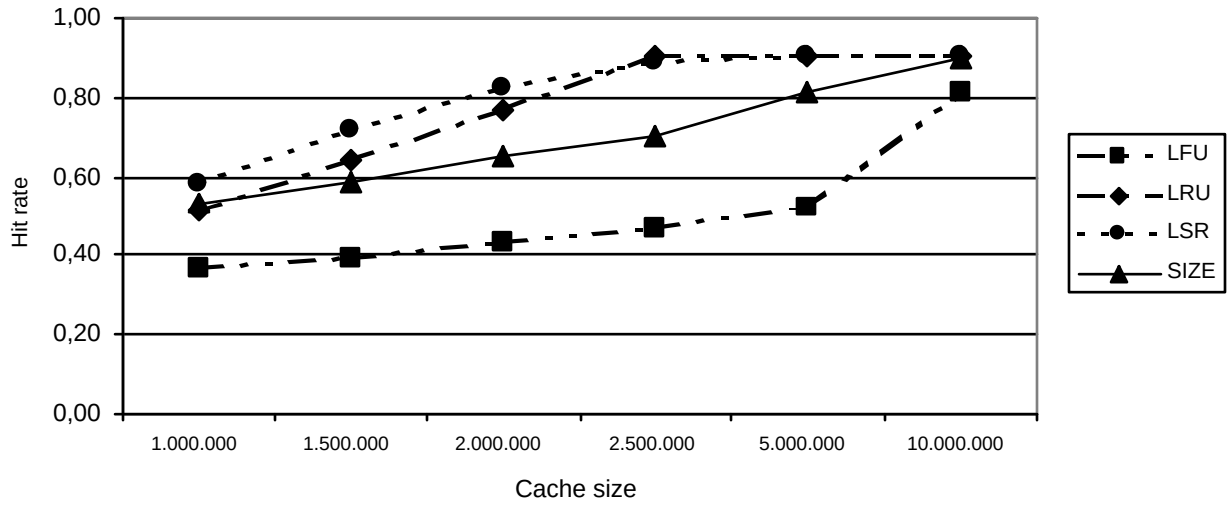

Figure 9: Hit rate for different cache sizes

Access Category	*Quantity of Objects*	*Probability*	*Range*	
			Lower Bound	*Upper Bound*
c	q_c	$P_c = \frac{q_c}{\sum_c q_c}$	$\sum_{i=1}^{c-1} P_i$	$\sum_{i=1}^{c} P_i$
1	20	0.47	0.00	0.47
2	10	0.23	0.47	0.70
3	6	0.14	0.70	0.84
4	4	0.09	0.84	0.93
5	2	0.05	0.93	0.98
6	1	0.02	0.98	1.00
Total	43	1		
	$\sum_c q_c$	$\sum_c P_c$		

Table 2: Monte Carlo Method probability distribution

pursuing a solution that applies the standard Vector Model widely employed by search engines for information retrieval. That will make LSR universally applyable on the Internet, independently of any taxonomy.

- Finally, LSR can be used for information pre-fetching rather than caching, especially because it considers users interests by construction.

Therefore, the main contributions of our research work is the proposal of a novel policy for object cache replacement, its implementation and validation, and a complete simulation model. The first results favor LSR – users can get their objects faster and the network is less loaded – and open new research perspectives, especially in the context of Semantic Web, where the assumed object semantics is a fact.

References

[Aggarwal et al., 1999] Aggarwal, C. C., Wolf, J. L., and Yu, P. S. (1999). Caching on the World Wide Web. *Knowledge and Data Engineering*, 11(1):95–107.

[Arlitt et al., 1998] Arlitt, M., Friedrich, R., and Jin, T. (1998). Performance evaluation of Web proxy cache replacement policies. *Lecture Notes in Computer Science*, 1469:193+.

[Atzmon et al., 2002] Atzmon, H., R.Friedman, and R.Vitenberg (2002). Replacement policies for a distributed object caching service. In *International Symposium on Distributed Objects and Applications (DOA)*.

[Brandão and Anido, 2001] Brandão, R. F. and Anido, R. d. O. (2001). Um simulador parapelo para sistemas de caches para web. In *Proceedings of the 19th Brazilian Simposium on Computer Networks*, pages 480–495.

[Cao and Irani, 1997] Cao, P. and Irani, S. (1997). Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA.

[Cheng and Kambayashi, 2000] Cheng, K. and Kambayashi, Y. (2000). Advanced replacement policies for WWW caching. In *Web-Age Information Management*, pages 239–244.

[Chidlovskii et al., 1999] Chidlovskii, B., Roncancio, C., and Schneider, M.-L. (1999). Semantic cache mechanism for heterogeneous web querying. *WWW8 / Computer Networks*, 31(11-16):1347–1360.

[Dar et al., 1996] Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proc. VLDB Conf.*, pages 330–341, Bombay, India.

[dos Santos, 2001] dos Santos, R. G. (2001). Internet object cache replacement based on semantics. Master's thesis, Pontificia Universidade Catolica do Parana, Curitiba, Brazil.

[Keller and Basu, 1996] Keller, A. M. and Basu, J. (1996). A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47.

[Liu, 2001] Liu, J. S. (2001). *Monte Carlo Strategies in Scientific Computing*. Springer Verlag. 360 pages.

[Murta et al., 1998] Murta, C. D., Almeida, V. A. F., and Meira Jr., W. (1998). Analyzing performance of partitioned caches for the WWW. In *Proceedings of the 3rd International WWW Caching Workshop*.

[O'Neil et al., 1993] O'Neil, E. J., O'Neil, P. E., and Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. In *Proceedings*

of the ACM SIGMOD International Conference on Management of Data, pages 297–306.

[Ren et al., 2003] Ren, Q., Dunham, M. H., and Kumar, V. (2003). Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):192–210.

[Rizzo and Vicisano, 2000] Rizzo, L. and Vicisano, L. (2000). Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170.

[Shmidt, 2002] Shmidt, G. (2002). Semantics server for internet resources: an approach based on rdf. Master's thesis, Pontificia Universidade Catolica do Parana, Curitiba, Brazil.

[Silberschatz and Galvin, 1994] Silberschatz, A. and Galvin, P. B. (1994). *Operating Systems Concepts*. Addison-Wesley, Reading, Mass., fourth edition.

[Vakali, 2001] Vakali, A. (2001). Proxy cache replacement algorithms: A history-based approach. *World Wide Web*, 4:277–298.

[Williams et al., 1996] Williams, S., Abrams, M., Standridge, C. R., Abdulla, G., and Fox, E. A. (1996). Removal policies in network caches for World-Wide Web documents. In *Proceedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, USA.

[Wooster and Abrams, 1997] Wooster, R. P. and Abrams, M. (1997). Proxy caching that estimates page load delays. *Computer Networks*, 29(8-13):977–986.

[Zheng et al., 2002] Zheng, B., Xu, J., and Lee, D. L. (2002). Cache invalidation and replacement strategies for location-dependent data in mobile environments. *IEEE Transactions on Computers*, 51(10):1141–1153.