Military Technical College
Kobry El-Kobbah
Cairo, Egypt

10th International Conference
On Aerospace Sciences&
Aviation Technology

A LOW LATENCY PROXY PREFETCHING CACHING ALGORITHM

Fatma A. Omara*
*Assoc. Prof., Computer Science Dept., Faculty of Computers and Information, Cairo University, IEEE Member, Fax 3350109.

Abstract

The Web proxy cache system was deployed to save network bandwidth, balance server load, and reduce network latency by storing copies of popular documents in the client and proxy caches for the Uniform Resource Locator (URL) requests. To solve the problem of the Web's slow end-user response time, a Web proxy caching and prefetching strategy has been developed and implemented by the auother to provide the users by the information they mostly likely want to browse in user profiles. This developed strategy uses the Reverse Aggressive technique for prefetching, which was proposed theortically. This developed strategy has been implemented with different cache sizes using a Web caching simulator. The tradional caching replacement policies such as Least-Recently-Used (LRU), Hybrid, and Size policies were already existed in this simultor. This simulator has been modified by the work in this paper such that the most recent replacement policies; Last-In-First-Out (LIFO), First-Try, Swapping and Place-Holder policies under infinite sized cache have been implemented. The performance measurements of the developed strategy have been studied using the tradional replacement policies, and the most recent replacement policies. Also, a comparative study has been done to clarify the benefits of the Reverse Aggressive caching prefetching algorithm with respect to the Fixed-Horizon caching prefetching algorithm with respect to the Reduced Latency (RL). According to the implementation results, it has been found that the average latency has been reduced at a higher degree by using the Reverse Aggressive cache prefetching strategy.

Index Terms – Caching, Proxy caching, File Systems, and Operating Systems

1. Introduction

Rapid improvements in the processor and memory speeds have created a situation in which I/O, in particular file system I/O, has become the major bottleneck to the operating systems performance. Unfortunately, access latency is still a problem and is not likely to improve significantly due to the physical limitations of storage devices and network transfer latencies [1].

During the last few years, using of the World Wide Web (WWW) is increased exponentially by increasing the number of users, both individuals and businesses, as well as increasing of servers, multimedia contents and real-time audio/video transmissions. In addition, the Internet pipes got bigger and bigger by using new technologies. This allows the users to access more data during the time they are connected to the Internet. In addition, the contents of the Web documents become more complicated where more multi-media capabilities have been included. These lead to increase the Hyper Text Transmission Protocol (HTTP) Traffic [2]. On the other hand, the HTTP traffic has the following Problems:

- Increase in the network bandwidth usage.
- Increase in the latency.
- Increase in the load overhead on document servers.

Therefore, caching at various points provides a natural way to overcome these Web problems. A common form of Web caching is at HTTP proxies. These HTTP proxies are intermediaries between browser processes (i.e., clients nodes) and Web servers on the Internet (see Fig. (1)).

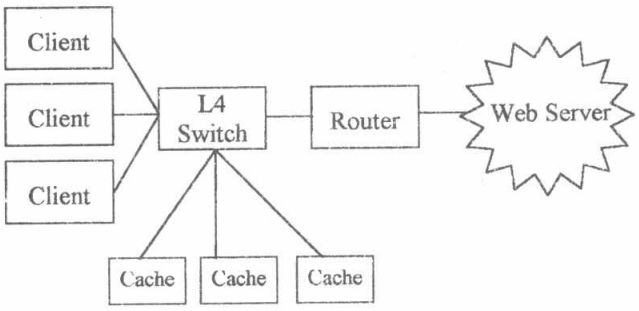


Fig. (1) The Common Proxy Caching Configuration

In general, the use of caching proxies can reduce three different issues [2]:

1. The number of requests that reach servers,
2. The volume of network traffic resulting from document requests,
3. The latency that an end-user experiences in retrieving a document.

On the other hand, the latency caused by clients or server could be reduced by using prefetching between caching proxy and browsers. Furthermore, the prefetching takes a heuristic-based approach using knowledge of past accesses to predict future access without user or application intervention. As a result, applications automatically receive reduced latencies, better use of available bandwidth batch file system via batched file system requests, and improved Web proxy cache utilization [1].

Although caching and prefetching are key ideas in the success of the Web, an inefficient forced approach to them will lead to disastrous performance. On the other hand, many strategies for prefetching in read-intensive applications have been used such as Fixed-Horizon, Aggressive, and Reverse - Aggressive strategies.

The Fixed-Horizon prefetching is based on the Second Informed Prefetching (TIP2) system of Hugo Patterson, Gibson et al.[3]. This system manages allocation of cache space and I/O bandwidth between multiple processes, where only some of which are disclosing some or all of their future accesses. TIP2 is designed for the case in which sufficient I/O bandwidth exists to service the request stream without stalling on I/O [4].

Aggressive prefetching prefetches a block as early as possible, provided that the prefetched block is needed by the application sooner than the block that it will replace. When insufficient bandwidth is available, in particular, it becomes more important to schedule prefetch requests to ensure that no bandwidth is wasted. It uses knowledge of future accesses to minimize application elapsed time for both small and large numbers of disks [5].

The Reverse-Aggressive prefetching decisions depend on information farther in the future than the other algorithms because it prefetches blocks in parallel if these blocks reside on different disks [6]. This strategy was proposed therotically, and it has been implemented and verficated by the work in this paper.

According to the work in this paper, the sophisticated caching and prefetching strategy using the Reverse-Aggressive technique with different replacement policies, has been developed to improve the Web performance without saturating the Internet with useless traffic. The developed caching and prefetching strategy has been implemented using the Web simulator [7]. This Web simulator includes steps to implement Fixed-Horizon caching and prefetching strategy using the tradional replacement policies; LRU, Size, and Hybrid. The simulator has been modified by the work in this paper such that the implementation of the Reverse-Aggressive caching and prefetching strategy has been included. Also, the most recent replacement policies; First-Try, Swapping Position, and Last-In-First-Out (LIFO), and Place-Holder policies are implemented and included. Furthermore, a comparative study of the Web caching prefetching strategy based on the Reverse-Aggressive technique and the Fixed-Horizon technique has been implemented to reveal the benefits of using the Reverse-Aggressive prefetching with caching using the tradional replacement policies, as well as, the most recent replacement policies

The rest of this paper is organized as follows; section 2 handles the different proxy caching policies. Section 3 handles the modification propagation schemes to over- come the cache inconsistent problem, which is considered very

important in the write applictions case. The caching infrastructure schemes are handled in section 4. Section 5 is dedicated to illustrate the Reverse-Aggressive prefetching technique implementation. Section 6 illustrates the used Web cach simulator. The performnce measurements of the proposed Web caching prefetching strategy using different replacement policies with respect to RL prameter are given in section 7. Also, a comparative study of the Web caching prefetching using Reverse-Aggressive technique and Fixed-Horizon technique using different replacement policies is presented in section 7. Finally, the conclusions are presented in section 8.

2. The Proxy Caching Policies

The behavior of a cache is defined by a set of cache policies such as prefetching, routing, coherency, and removal policies [8].

2.1 Prefetching Policies

Prefetching can be used with the Web proxy caching in different ways. For example, a client might use prefetching to load objects from the cache into the browser. This may be interesting in reducing the latency due to slow dialup connections. On the other hand, caches could be customized to accept objects pushed out from servers. This provides a technique whereby servers can perform push-caching of popular documents, and as a result, fewer requests will be served at the origin server reducing server connection and improving performance [8].

2.2 Routing Policies

The routing policies determine how a cache retrieves an object. Clients, caches and servers might use the routing policy in a different manner. Clients could specify routing policies that use different prior services. A cache administrator, on the other hand, might specify a routing policy that allows cooperation among multiple caches. A routing policy provided by a site might be used to alternate requests between different mirrors of that site. Even better, the routing policy might determine the optimal mirror the cache should contact [8].

2.3 Coherency Policies

The coherency policies determine how a cache responds to a request for an object in the cache. The coherency policy decides either to consider the object fresh or stale. In the case of stale objects, the cache consults the server to verify that the copy is up-to-date. A commonly used algorithm in deciding coherency is the TTL algorithm [9]. This algorithm determines whether an object is fresh or stale by evaluating the following equation:

$$T_{now} - T_{in} < K\,(T_{in} - T_{last\text{-}mod})$$

Where $\mathbf{T_{now}}$ is the time of the request, $\mathbf{T_{in}}$ is the time when the object entered the cache, $\mathbf{T_{last\text{-}mod}}$ is the time when the object was last changed, and K is a constant. If the Boolean expression is true, then the document is considered fresh [9].

2.4 Removal Policies

Two considerations must be taken into account in designing a removal policy for a Web caching system. First consideration is that real caches have finite size, and thus a request for a non-cached document may cause the removal of one or more cached documents. Second consideration is related to the effect of removal policies on Hit Rate (HR), Weighted Hit Rate (WHR), and cache size parameters. On the other hand, the term "removal" policy is used rather than "replacement" because a policy might be run periodically to remove documents until free space reaches a threshold. Removal polices are based on factors such as document size, time the document entered the cache, time of last access, number of references, in deciding which document has to be removed [10].

3. Modification Propagation Schemes

In the file systems in which the cache is located on clients' nodes, a file's data may simultaneously be cached on multiple nodes. In such a situation, when the caches of all these nodes contain exactly the same copies of a file data, then the caches are consistent. It is possible for the caches to become inconsistent when the file data is changed by one of the clients and the corresponding data cached at other nodes are not changed. The used modification propagation scheme has a critical effect on the system's performance and reliability. Two schemes can be used to modify propagation [11]. These are write-through and delayed-write schemes.

3.1. Write-Through Scheme

According to the write-through scheme, when a cache entry is modified, the new value is immediately sent to the server for updating the master copy of the file. This scheme has two main advantages; high degree of reliability and suitability since every modification is immediately propagated to the server having the master copy of the file. The risk of updated data getting lost (when a client crashes) is very low. A major drawback of this scheme is its poor write performance because each write access has to wait until the information is written to the master copy of the server. Notice that with the write-through scheme, the advantages of data caching are only for read accesses. Therefore, this scheme is suitable for use only in those cases in which the ratio of read-to-write accesses is fairly large [11].

3.2. Delayed-Write Scheme

Although the write-through scheme helps on reads, it does not help in reducing the network traffic for writes. Therefore, to reduce network traffic for writes as well, some systems use the delayed-write scheme. In this scheme, when a cache entry is modified, the new value is written only to the cache and the client just makes a note that the cache entry has been updated. Some time later, all updated cache entries corresponding to a file are gathered together and sent to the server

at a time. Different policies are used with delayed-write scheme. The most commonly used approaches are as follows:

Write on Ejection From Cache;

According to this method, modified data in a cache entry is sent to the server when the cache replacement policy has decided to eject it from the client's cache. This method can result in good performance, but some data can reside in the client's cache for a long time before they are sent to the server. Such data are subject to a reliability problem [12].

Periodic Write;

In this method, the cache is scanned periodically, at regular intervals, and any cached data that have been modified since the last scan are sent to the server. Sprite [12] uses this method with an interval of 30 seconds.

Write On Close;

In this method, the modifications made to a cached data by a client are sent to the server when the corresponding file is closed by the client. Notice that the write-on-close policy is a perfect match than others. However, it does not help much in reducing network traffic for those files that are open for very short periods or are rarely modified. Furthermore, the close operation takes a long time because all modified data must be written to the server before the operation completes. Therefore, this policy should be used only in cases in which files are open for long periods and are frequently modified [9].

Furthermore, the delayed-write scheme helps in performance improvement for write accesses due to the following reasons [12]:

- Write accesses complete more quickly because the new value is written only in the cache of the client performing the write.
- Modified data may be deleted before the time to sent them to the server. For example, many programs create temporary files, use them, and delete them soon after they are created. In such cases, modifications need not be propagated at all to the server, resulting in a major performance gain.
- Gathering of all update files then sending them together to the server is more efficient than sending each update separately.

However, delayed-write scheme suffers from the reliability problem, since modifications not yet sent to the server from client's cache might be lost if the client crashes. Another drawback of this approach is that delaying the propagation of modifications to the server results in fuzzier file sharing, because when another process reads the file, what it gets depends on the timing [12].

4. The Caching Infrastructure

The caching infrastructure is based on subdividing a policy into actions that are taken when specific events occur [8]. In Fig. (2), the relevant states of a Web

object are shown. The arcs in the figure represent events that cause an object state to change [8].

For example, the changes in state of the Web object http://wonderland.net/tea-party.html, are illustrated; firstly the Dormouse requests the tea party object and later Alice does. Initially, the tea party object exists in the **Not Cached** state. When the Dormouse makes a request for the tea party object, the routing policy is used to determine how the cache retrieves a copy of the tea party object. This object enters the **Private Object** state. Now, the placement policy determines whether the tea party object is stored in the cache or not. If the object is stored, then it enters the **Fresh Public Object** state. Later, when Alice requests the tea party object, the **Requested** state is entered. At this state, the coherency policy determines whether the object is fresh or stale and, if stale, the coherency policy checks whether the original object has changed using a **Get If-Modified-Since**

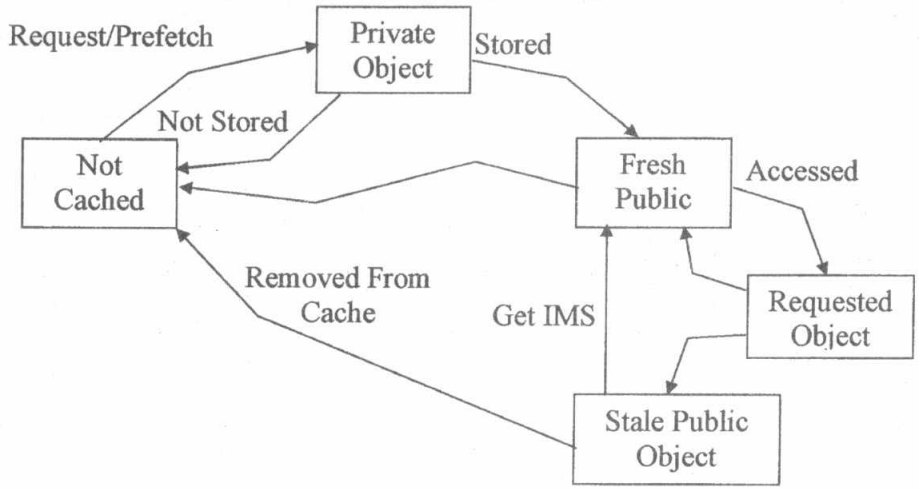


Fig. (2) States of a Document in a Cache

(IMS) request [8]. The amount of space available in the cache also affects the state of Web objects. When the cached bytes exceed capacity, the cache uses a removal policy to determine the best objects to be removed. These objects leave the **Fresh** or **Stale Public Object** states and return to the **Not Cached** state [8]. The work in this paper considers the object in the case of prefetching and storing in the cache (i.e., **Private** Object, **Fresh Public**, and **Stale Public** states).

Although there are prefetching techniques like Fixed-Horizon, Aggressive, and Reverse-Aggressive, we focus on the Reverse-Aggressive prefetching technique, which was proposed theoretically, but has been implemented using a Web simulator by the work in this paper.

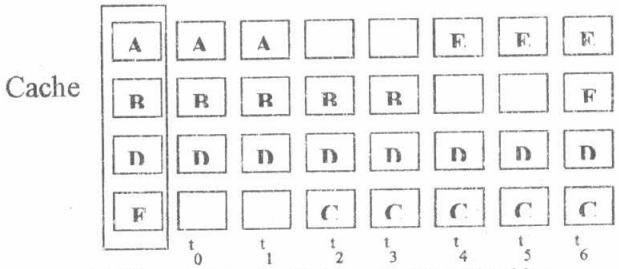
5. The Reverse-Aggressive Technique

It is a theoretical model that captures the important characteristics of a system for prefetching and caching from multiple disks. The optimal prefetching and caching schedule in this model minimize the elapsed time required to serve a given request stream [6]. To illustrate this, consider the following example.

Example:

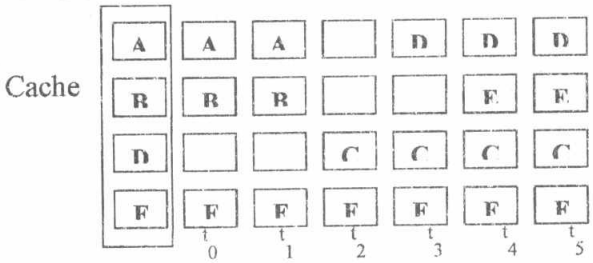
Consider an application that references blocks in sequence (A, B, C, D, E, F), and the cache initially holds A, B, D, and F. Blocks A, C, E, and F reside on one disk; blocks B and D on a different disk (see Fig. 3). A straight forward approach is to use the aggressive algorithm [5] which always fetch the missing block that will be referenced sonnets, and evicts the block whose next reference is furthest in the future, but do not fetch if the evicted block will be referenced before the fetched block. Fig. (3.a) represents the cache block changes. The produced schedule by this algorithm stalls for one unit of time. Another schedule is represented in Fig. (3.b), which is considered faster by one time unit. On the first fetch, D is evicted rather than F, even though D is referenced earlier. This allows two fetches to proceed in parallel (i.e., reverse aggressive is used) , thus saving one time unit.

References	A	B	C	D	E	F
Prefetches	C		E		F	
Evictions	F		A		B	



(a) A Natural Extension To The Aggressive Algorithm.

References	A	B	C	D	E	F
Prefetches	C	D	E			
Evictions	F	A	B			



(b) Better Algorithm

Fig. (3) Prefetching and Caching Using Two Disks

This example shows that it is helpful to take disk load into account when making fetching and eviction decisions. This is the factor that makes the multi-disk problem more difficult than the single-disk problem [6]. The advantages of the Reverse-Aggressive Algorithm over the Aggressive one are [6]:

1. Whereas aggressive algorithm chooses evictions without considering the relative loads on the disks, reverse aggressive algorithm evicts as many disks as possible on the reverse sequence. This results in performing close to the best evictions possible for those fetches.
2. Whereas aggressive algorithm can wastefully prefetch ahead on some of its disks, reverse aggressive algorithm is in the reverse direction. In the forward direction, this translates to performing a maximal set of fetches in parallel.

6. Web Cache Simulator

The used Web cache simulator uses pre-processed Web access traces, which are gathered and published by the Computer Science Department of Boston University and the Digital Equipment Corporation (DEC) in the USA, respectively [7]. The inputs of these trace files for each HTTP request are as follows:

Unique Web Server ID: It is a unique integer number, which specifies the Web server address of the request.

Unique Path ID: It is a unique integer number, which specifies the path of the request.

File Size: It is the size of the accessed document, in bytes.

Latency: The duration of the HTTP request in microseconds.

Last Modified Time: It is the last modified timestamp in the HTTP reply header.

Access Time: The time stamp of the HTTP request in seconds.

This Web simulator does consider cooperation between caches as well as concurrent caching operation. The cache operations allow scripts to store objects in the cache and purge objects to make more space. Internally, the simulator maintains statistics about true/false hits, true/false misses, and latencies. A true hit occurs when the object being requested hits in the cache and the object has not been modified from the last version stored in the cache. This simulator is used to simulate the traditional replacement policies like LRU, Size, and Hybrid. It is modified by the work in this paper such that the most recently replacement policies; LIFO, First-Try, Swapping and Place-Holder have been included. Because this simultor considers proxy cache with different sizes, then the Fixed-Horizon prefetch strategy is already considered. According to the work in this paper, the Reverse-Aggrassive prefetch strategy has also been implemented in the simulator by adding a buffer with specific size to store the references, which will be prefetching without waiting to store them in the cache. That is the opposit of what happens in the Aggressive strategy, where there is a specific

time that must be included after each prefetch to make sure that the reference is already stored in the cache.

7. The Performance Measurements

A comparative study has been done to compare the Reverse-Aggressive and the Fixed-Horizon caching prefetching strategies in reducing latency using a Web proxy caching simulator using the traditional replacement policies; LRU, Size, and Hybrid; as well as; the most recent replacement policies; First-Try, Swapping, Place-Holder and Last-In-First-Out (LIFO) policies.

According to **LRU** policy, the document that was least recently requested will be selected to be evicted. According to **Size** policy; the large objects from caches will be selected to be evicted to reduce bytes requested from server and client latencies [6]. These policies depend on a single attribute of the object. For instance, **Size** removes large objects from the cache, which make room for multiple smaller objects. So, it improves the hit rate, but may degrade client latencies due to the high network overhead of uncached large documents, As a result, cache's hit rate improves. According to **Hybrid** policy, the evicted decoument will be selected according to the following equation [13].

$$\frac{\left\{C_S + \frac{W_B}{B_S}\right\}(N_P)^{W_N}}{Z_P}$$

i.e., the operation for a document **P** located at server **S** depends on the following parameters:

Cs; the connected time with server S,

Bs; the bandwidth of the servers,

Np; the number of times P has been requested since it was brought into the cache,

Zp; the size of the document **P** (in bytes).

WB and **WN** are constants.

The estimated values for **Cs** and **Bs** are based on the time to fetch documents from server **S** in the recent past [14]. The document with the smallest calculated value will be evicted.

7-1 The Recent Replacement Policies

There are four polices have been recently developed, First-Try, Swapping, Place-Holder, and Last In First Out (LIFO) algorithms.

According to **LIFO** policy, the first block in the list is the one to be evicted. The **First-Try** policy considered the global **LRU** policy, where the kernel maintains a LRU list of all blocks currently in the file cache. When a replacement is necessary, the block at the end of the LRU list is suggested as a candidate, and its owner process is asked to give up a block [15,16]. The problem of this policy

is that if the owner process over-rules the kernel, the candidate block still stays at the end of the LRU list. On the next miss, the same process will again be asked to give up a block.

The pseudo code of the **First-Try** policy is as follows:

```
/* First-Try Algorithm */
if (inf.)
 {
   /* LRU-policy */
   if (head → backward != Null)
   {
      head = head→ backward;
      free (head→ forward);
      head→ forward = Null;
   }
   else
   {
      free (head);
      head = Null;
      tail = Null;
   }
  }
 else
 {
   /* LIFO Policy */
   if (tail→ forward! = Null) {
   tail = tail→ forward;
   free (tail→ backward);
   tail→ backward = Null;}
   else {
   free (tail);
   head = Null;
   tail = Null;}}
```

The problem of the First-Try policy arises because the LRU list is maintained in strict reference order. Intuitively, the only use of the LRU list is to decide which process will give up a block upon a miss. To get the same allocation policy as the existing global LRU policy, the suggested policy in the First-Try policy should be in correspondence to the LRU list in the original algorithm. This can be achieved by swapping the blocks' positions in the LRU list. This is what happens in the Swapping policy.

The **Swapping** policy guarantees that if no process makes foolish choices, the global hit ratio is the same as or betters than it would be under the global LRU policy [15,16]. The psoudo code of the **Swapping** is as follows:

```
/* Swap Algorithm */
    { if (same_pageID (head→ page, rem→ page)) {
            head= head→ backward;
            head→ forward = Null;}
             if (rem →  forward != Null)
               rem→ forward→ backward = rem→
               backward;
              if (rem → backward != Null){
                rem→ backward→ forward = rem→
                forward;
                rem→ forward = tail;
                tail→ backward = rem;
                tail = rem;
                rem→ backward = Null } }
```

The **Place-Holder** is a record that refers to a page. It records which block would have occupied that page under the global LRU policy. Suppose the kernel suggests **A** for replacement, and the user process over-rules it and decides to replace **B** instead. In addition to swapping the positions of **A** and **B** in the LRU list, the kernel also builds a Place-Holder for **B** to point **A**'s page. If **B** is later referenced before **A**, **A**'s page can be confiscated immediately. Hence, the Place-Holder algorithm keeps tracking the differences between the replacement choices made by an application's policy and these that would have been made by the default LRU policy. The Place-Holder algorithm allows the system to detect when the application's choice is not as good as that of the default policy. In this case the erroneous process will give up a block [15,16]. The psoudo code of the **Place-Holder** policy is as follows:

```
                    /* Place-Holder Algorithm */
    if (same_pageID (page, ph→ page)){
      Ph→ forward = tail;
      Ph→ backward = ph;
      Tail =ph;
      Ph→ backward = Null; }
```

7.2 The Implementation Results

The implementation results of the Fixed-Horizon Cashing Prefetching for the proxy cache using different replacement policies with respect to the Latency Reduction (LR) are listed in tables [1]. Cache size S is in Bytes, and LR in m.sec.

Table 1. The Latency Reduction (LR) For The Fixed-Horizon Caching Prefetching Using Different Replacement Policies

Rep. Algorithms	S1	S2	S3	S4	S5	S6	S7
LRU	0.25256	0.27170	0.31900	0.37817	0.49808	0.97219	1.0000
Size	0.04054	0.05818	0.22108	0.34289	0.47597	0.89444	1.0000
Hybrid	0.00830	0.40329	0.55211	0.64972	0.87414	0.925	1.000
LIFO	0.03547	0.13547	0.48075	0.64972	0.87414	0.925	1.000
First-Try	0.26183	0.26183	0.25354	1.13194	0.95505	0.80093	1.55312
Swapping	0.46502	0.79733	0.75464	0.75463	0.96487	1.11521	1.55312
Place-Holder	0.465	0.797	0.754	0.754	0.964	1.115	1.553

The comparison results are shown in Fig. (4). It is found that by using the Swapping, Place-Holder replacement policies with the caching and prefeteching strategy such that the latency reduction can be satisfied.

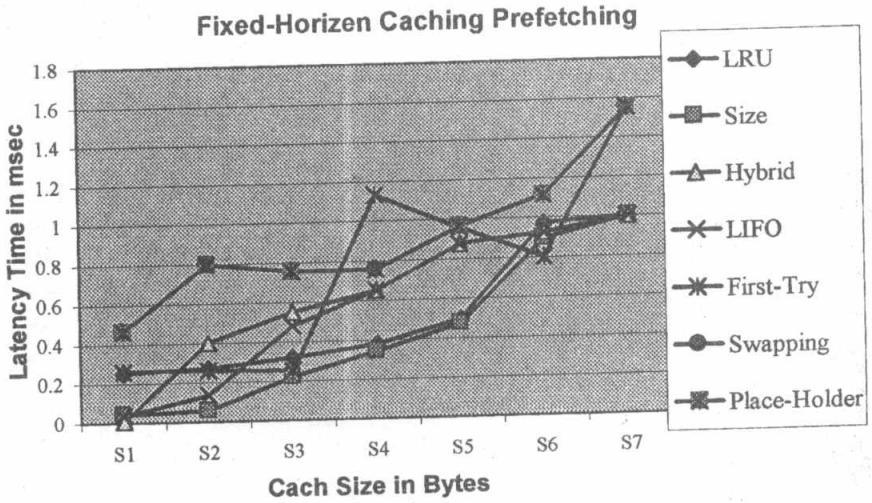


Fig. (4) The Fixed-Horizon Caching Prefetching For The Proxy Cache using Different Replacement Policies

The implementation results of the Reverse-Aggressive Cashing Prefetching for the proxy cache using different replacement policies are listed in tables [2].

Table 2. The Latency Reduction (LR) For The Reverse-Aggressive Caching Prefetching Using Different Replacement Policies

Rep. Algorithm	S1	S2	S3	S4	S5	S6	S7
LRU	0.26183	0.45354	0.95505	0.80093	1.55312	3.29971	3.39409
Size	0.13760	0.16308	0.44082	0.89243	1.33393	3.36162	3.39409
Hybrid	0.11417	0.93675	0.62106	2.04406	2.63935	3.34120	3.39409
LIFO	0.0364	0.42825	0.49315	1.13194	2.63935	3.34120	3.39409
First-Try	0.26183	0.26183	0.25354	1.13194	0.95505	0.80093	1.55321
Swapping	0.46502	0.79733	0.75464	0.75463	0.96487	1.11521	1.55312
Place-Holder	0.465	0.754	0.754	0.754	0.964	1.115	1.553

The comparison results are shown in Fig. (5). According to these results, it is found that the LIFO, Size, LRU and Hybrid replacement policies are the best algorithms that could be used with the caching and prefeteching strategy such that the latency reduction can be satisfied.

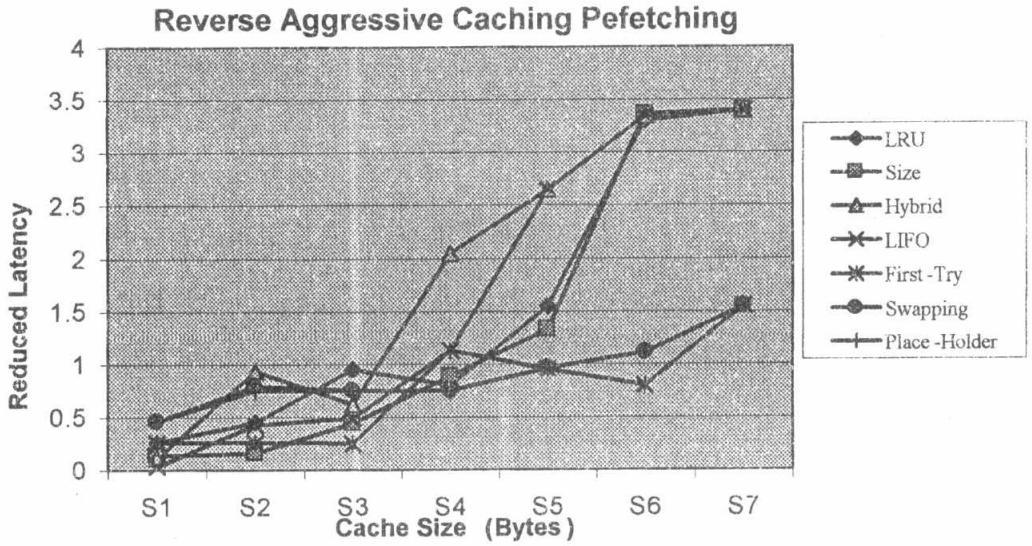


Fig. (5) The Reverse-Aggressive Caching Prefetching For The Proxy Cache using Different Replacement Policies

The implementation results of the Fixed-Horizon and the Reverse-Aggressive caching prefetching strategies using different replacement policies are listed in tables 3-9 using different cache sizes. The comparison results between the Fixed-Horizon and the Reverse-Aggressive are represented in Figs. (6-12).

Table 3 –The Latency Reduction (LR) using LRU Replacement Policy

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressiv	0.26183	0.4535	0.9550	0.80093	1.55312	3.29971	3.39409
Fixed-Horizon	0.25256	0.2717	0.319	0.37817	0.49808	0.97219	1

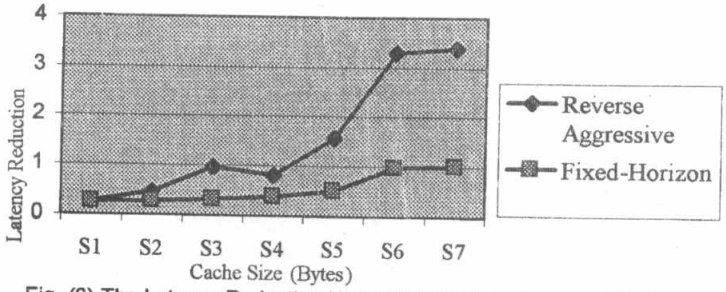


Fig. (6) The Latency Reduction Under The LRU Replacement Policy

Table 4 –The Latency Reduction (LR) using Size Replacement Policy

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.1376	0.1630	0.4408	0.8924	1.3339	3.3616	3.3940
Fixed-Horizon	0.0405	0.0581	0.2210	0.3428	0.4759	0.8944	1

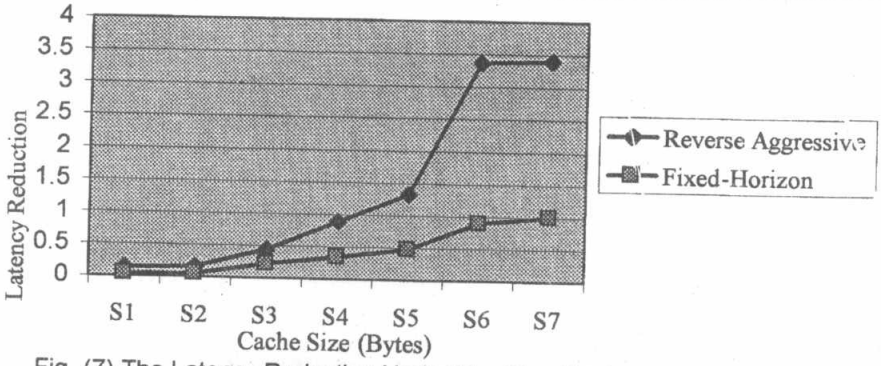


Fig. (7) The Latency Reduction Under The Size Replacement Policy

Table 5 –The Latency Reduction (LR) Using The Hybrid Replacement Policy

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.11417	0.93675	0.62106	2.04406	2.63935	3.34120	3.39409
Fixed-Horizon	0.0083	0.40329	0.55211	0.64972	0.87414	0.925	1

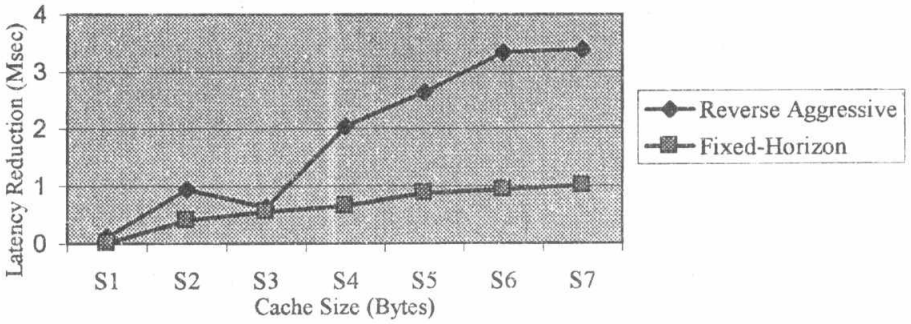


Fig. (8) The Latency Reduction Under The Hybrid Replacement Policy

Table 6 –The Latency Reduction (LR) Using LIFO Replacement Policy

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.0364	0.4282	0.4931	1.1319	2.6393	3.3412	3.3940
Fixed-Horizon	0.0354	0.1354	0.4807	0.6497	0.8741	0.925	1

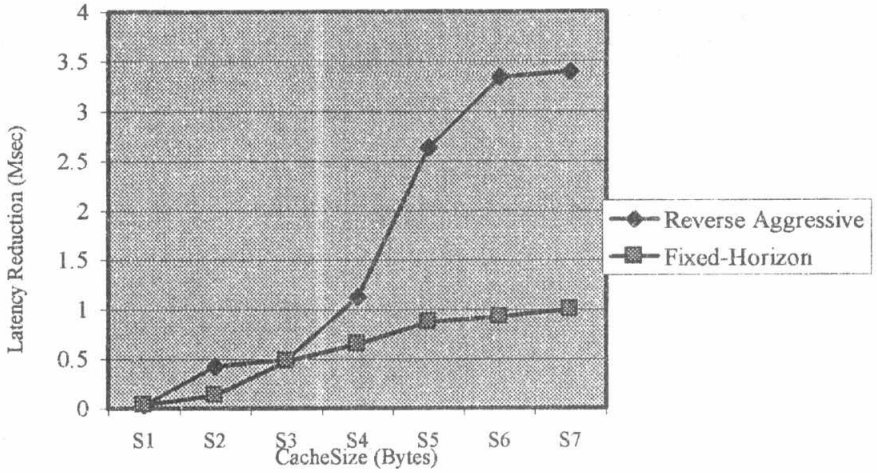


Fig. (9) The Latency Reduction Under The LIFO Replacement Policy

Table 7 –The Latency Reduction (LR) Using First - Try Replacement Algorithm

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.2618	0.2618	0.2535	1.1319	0.9550	0.8009	1.5532
Fixed-Horizon	0.2618	0.2618	0.2535	1.1319	0.9550	0.8009	1.5531

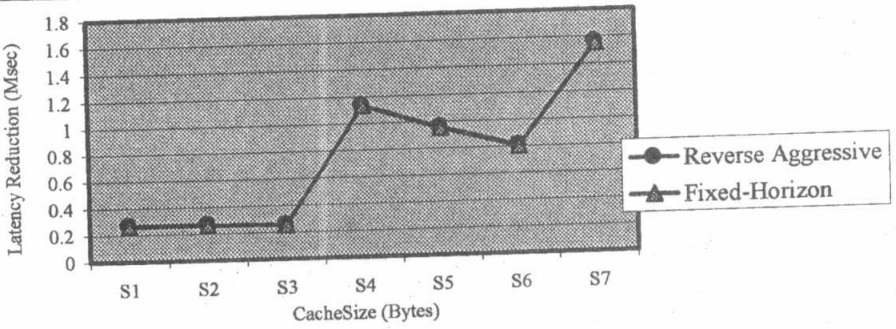


Fig. (10) The Latency Reduction Under The First - Try Replacement Policy

Table 8 –The Latency Reduction (LR) Using Swapping Replacement Algorithme

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.4650	0.7973	0.7546	0.7546	0.9648	1.1152	1.5531
Fixed-Horizon	0.4650	0.7973	0.7546	0.7546	0.9648	1.1152	1.5531

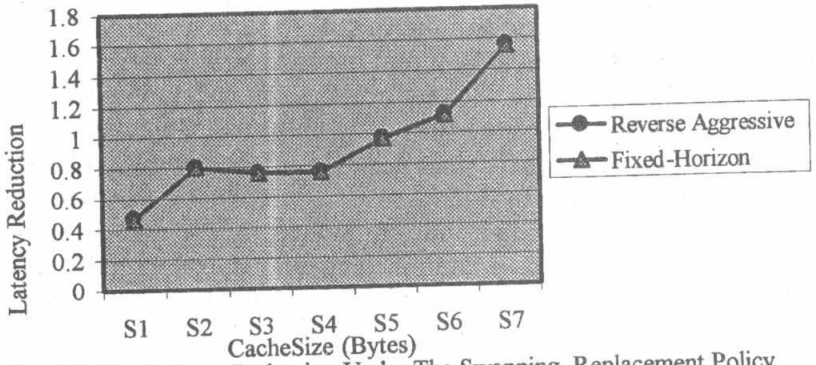


Fig. (11) The Latency Reduction Under The Swapping Replacement Policy

Table 9 –The Latency Reduction (LR) Using Place-Holder Replacement Policy

	S1	S2	S3	S4	S5	S6	S7
Reverse Aggressive	0.46	0.79	0.754	0.75	0.96	1.115	1.55
Fixed-Horizon	0.46	0.79	0.754	0.75	0.96	1.115	1.55

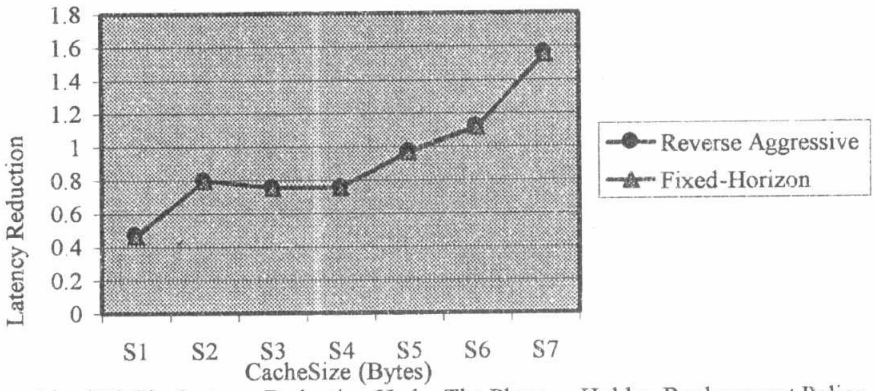


Fig. (12) The Latency Reduction Under The Place - Holder Replacement Policy

It is clear that by using the Reverse-Aggressive caching prefetching strategy, the average latency has been reduced compared with the Fixed-Horizon strategy.

8. Conclusions

The effectiveness of prefetching depends on how well the prefetching policy can predict which objects will be accessed and therefore should be retrieved in advance. The Reverse-Aggressive technique, which is probably close to optimal, is a perfect match than other policies like the Fixed-Horizon. Using a proxy caching simulator to implement and produce performance measurements of a file access, we compare the performance of using the Reverse-Aggressive cashing prefetching strategy with Web caching using the traditional replacement policies; LRU, Size, and Hybrid and the most recently replacement policies; LIFO, First-Try, Swapping and Place-Holder assuming an environment in which a single process is running and full advance knowledge is available. Also, a comparative study has been implemented between the Reverse-Aggressive caching prefetching strategy and the Fixed-Horizon cashing prefetching strategy. Based on the comparison results, we have recommended that if the proxy wants low average latency, the Reverse-Aggressive cashing prefetching strategy is the best policy under small cache sizes

References

1. Griffioen James, and Appleton Randy, "Reducing File System Latency Using A Predictive Approach," TR #CS247-94, June 1994.

2. Gracias A. A and Sivakumar G., "Seminar Report on World Wide Web Caching," Computer Science and Engineering Department, November 2000, Private Communication.

3. Hugo R. Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka, "Informed Prefetching and Caching," in The of the 15th ACM Symp. On Operating System Principles, Vol. 3, No. 6, PP. 79-95, December 1995.

4. Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brain Bershad, Pei cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and kai Li, "A Trace-Driven Comparison Of Algorithms For Parallel Prefetching And Caching," In The Proceedings Of The USENIX 2nd Symposium On Operating Systems design and Implementation (OSDI 96) Seattle Washington, PP. 19-34, October 1996.

5. Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li, " A Study Of Integrated Prefetching And Caching Strategies," SIGMETICS 95, Ottawa, Ontario, Canada, ACM, Vol. 23, No. 1, PP. 188-197, May 1995.

6. Kimbrel Tracy, and Anna R. Karlin, " Near-Optimal Parallel Prefetching And Caching," IEEE Symposium on Foundations of Computer Science, Vol. 14, No. 16, PP. 540-549, October 1998.

7. http://www. WebSim-A Trace Driven Web Cache Proxy Simulator.html.

8. Fritz J. Barnes, Pandey Raju, "Providing Dynamic and Customizable Caching Policies," in The Proceedings of the 2nd Symposium on Internet Technologies & Systems (USITS), October 1999.

9. Vincent Cate, " Alax-A Global File System," In USENIX File Systems Workshop Proceedings, PP. 1-12, May 1992.

10. Williams S., Abrams M., Standridge C. R., Abdulla G. and Fox E. A., "Removal Policies in Network Caches for World Wide Web Documents," Proceedings of The ACM Sigcomm 96, August 1996.

11. Pradeep K. Sinha, Distributed Operating Systems (Concepts and Design), IEEE Distributed Computing, January 1997.

12. Satyanarayanan et al., "A Caching File System for A Programmer's Workstation," Proceedings of the 10th ACM Symposium on Operating systems Principles, Vol. 4, No. 14, PP. 26-35, October 1980.

13. Satyanarayanan, Howard et al., "Towards A Scalable Metacomputing Storage Service-Pattern," Proceedings of the ITC Distributed File System: Principles and Design, 1997.

14. Wooster R. and Abrams M., "Proxy Caching That Estimates Page Load Delays," In The 6th International World Wide Web Conference, PP. 7-11, April 1997.

15. Cao P., filten E. W., and Li K., "Application-Controlled File Caching Policies," USENIX Summer workshop, PP. 171-182, June 1994.

16. Cao P., "Implementation and Performance of Application-Controlled File Caching, Prefetching and Disk Scheduling," Proceedings of The First USENIX Symposium on Operating Systems Design and Implementation (OSDI), PP. 165-178, November 1994.