RPC-Based Proxy Server

Assia Benbihi

Abstract

This paper sums up how a well set caching policy can drastically increase the performance of a web proxy server in term of cache hit rate and servicing time. The experiments show that the choice of the policy must be done according to the request distribution pattern and the cache memory size. A policy may perform more or less depending on these. In this project, we show that caching decision are optimum when they take into account long term benefits given the experiment environment. To do so, we compare four web proxy servers performance: one without caching policy, and three with respectively a RANDOM, LRU and LRU-MIN caching replacement strategy. Results show that LRU-MIN, i.e. the strategy that aims at long term high performance performs better than the other two in terms of hit rate and time of service. Then, LRU and RANDOM globally perform the same way in this experiment while RANDOM proves to be less sensitive to the workload pattern than LRU.

1. Introduction

Web services rely on server distributed all over the world to serve client requests. To do so, they must answer to data request at the client's request frequency. As more people make request to the same server, the servicing time increases which leads to increasing latency for the client.
To face this problem, a natural solution is to use a "divide and conquer" strategy on the server: distribute the server in sub-servers with each answering only some requests.
Then comes the problem of how to manage these servers: which server must serve which request? Which server must serve this specific client? Which server the client must sent the request to? … There exist many server architectures depending on the specificity of the requests and the clients. Still, one that may be intuitive is to place a sub-server called proxy server near groups of clients that have a lot of requests in common. That way we can set this proxy server to serve these particular requests with optimum performance. There still exist the question on how to manage data storage to serve client requests correctly while optimizing the memory usage. That is when a relevant caching policy on the proxy server can help improve time performance of the service.
We apply this general idea to the particular case of web server. We first recall how the server and the proxy server interacts with the client in section 1, three caching policies in section 2 and section 3 details their implementation for the experiments. Section 4 and section 5 respectively presents the experiments and the result analysis.

2. Web server

2.1 Remote web server

A machine, called "client", wants to get the content of a web page from a remote machine called via a remote procedure call. If there is only one server, this server must keep all the web pages in its memory and answer many synchronous request leading to latency as the server cannot serve two requests synchronously.

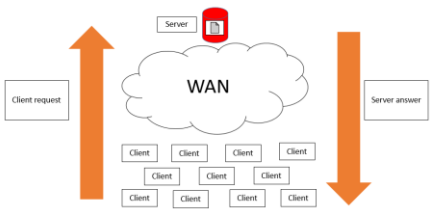


Figure1: Client-Server model

Several problems may occur in such a model such as latency issues depending on the distance and the type of network between client and server, contention issues depending on the level of demand to the server. A first approach to deal with these problems is to apply a “divide and conquer” strategy through proxy servers.

2.2 Proxy web server

Rather than locating the needed page in one server, the server is going to distribute copies of the content to “children” servers closer to the client. We will not study the coherence issues of such structure.

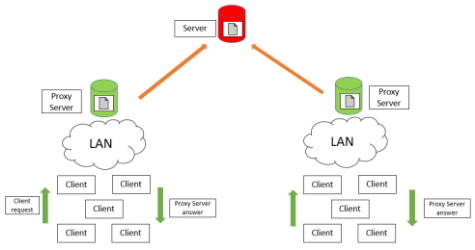


Figure2: Client- Proxy Server model

That way, next time a client want to get the page, it will ask the page to the proxy server rather than the original server. For the same number of clients, the number of requests per server decreases and so do the service latency.

There is still the problem of page storage management on the proxy server side. Since the proxy server serves only a part of the client pool, it may not need to keep all the pages in the remote server. From a design point of view, this is usually avoided due to the memory cost. Then there must be a trade-off between minimizing the size of stored web pages so that they fit in memory and maximizing the number of stored web pages so that the proxy can serve the client without requesting the page from the remote server. In that was the case, we would get back to the previous situation.

If the server does not store all the pages, he should then proceed as in Figure 3 when he receives a client request.

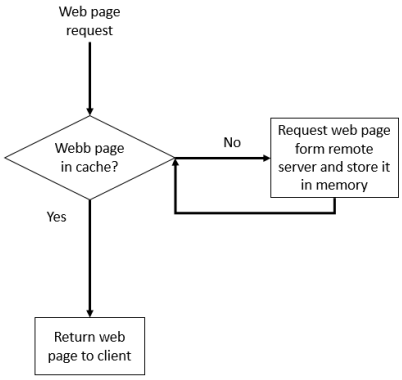


Figure3: General proxy sever algorithm

The first problem that arises is that since the cache memory size is limited, what to do when there is no space to store the page? Replacement is solution that appears naturally since the client requests are not static. So a page that was previously useful at some time, may not be useful anymore in the future.

The second problem is now to know which replacement policy to adopt. There exist several such policies called caching policies and each one is more or less relevant and performant given the client's request pattern and the cache memory size.

We have seen that the performance of the web service depends on the adopted caching policy. In the rest of this paper we focus on the study of three policies: RANDOM, LRU, LRU-MIN.

3. Cache replacement strategy

As we saw in Section 2, the proxy server must choose the caching policy to perform the minimum remote server request since this has high latency in the web service. There exist many caching policies and each one may prove better performance depending on the server constraints. Still, this project restricts the study to the following three caching policies: random policy (RANDOM), Least Recently Used (LRU), Least Recently Used (LRU-MIN).

3.1 Random replacement

This policy implements a brain-dead replacement: when the proxy needs to make room for a new web page, he randomly chooses pages to evict until there is enough room for the new one. The drawback of such a policy is that it treats equally the least used and most requested pages which can be critical for some situations.

For example if the cache evicts one of the most used page, once a client makes a call for this page, it will have to go through the WAN to get the page from the main server. Since this page is one of the most used, the probability of such a call is high and so do the expectation of the service latency. These calls could have easily been avoided by evicting a page that has low probability of being called.

On the contrary, this policy may be relevant if all the pages have equal probability of being called. Indeed, it would cost more time to implement a replacement policy than randomly picking a page from memory. But most of the times, web requests follows patterns depending on their geographical regions, time ... In such case, the distribution of requested pages will not be uniform and it is necessary to use another caching policy.

3.2 LRU strategy

The distribution of requested web page may be fix over time: for example, we make recurrent requests to http://google.fr. So it may be relevant to set a caching policy that favours the most frequently used web pages. However such strategy will imply that any "outlier" requested web page will automatically generates high latency because it has no chance of being in the cache due to its lack of popularity. If this page becomes one that is popular among clients, the latency due to its servicing will not decrease until it becomes as popular as Google.

To promote equality among web pages, the LRU policy favours the most recently used pages: when the proxy needs to evict a page from memory, it looks for the least recently used pages and evicts them until there is enough room for the new one.

This policy assumes that pages that have recently been requested have high probability of being requested again which is proved to be true in practice. Still, this strategy, just as the random one, does not take eviction decisions based on long term benefits. Indeed, imagine that the proxy server received only web request for pages of increasing "small sizes" until now. Suddenly, an outlier page i.e. a "big page" is requested only once by a client. Both strategies are going to evict a maximum number of pages : RANDOM chooses randomly among small pages so it has no choice but repeating enough eviction until there is enough room. LRU evicts the least recently used pages that are the smallest one in this

particular case. Then, when the "small pages" will be requested, the cache will face only misses which leads to high request latency.

This shows that the proxy must think of long term performance regarding the request latency and that is what the LRU-MIN strategy tries to implement.

3.3 LRU-MIN strategy

The LRU-MIN strategy tries to minimize the number of eviction by taking into account both the last time the page was requested and its size. When the proxy must make room for a page of size S, it looks for the pagess of size $T \geq S_0 = \frac{S}{2}$ and evicts the least recently used until there is enough room for the new page. If there is not enough room yet and there are no more page, it does the same with the pages $T \geq S_1 = \frac{S_0}{2}$ and keeps going until there is enough room.

If we examine the previous situation, we see that the number of eviction is lower that for the other strategy. Since it evicts first the larger least recently used pages, it will make room for the new page using less eviction than by evicting many little pages. Then after this "outlier request", the proxy will face less miss and the latency request will not be as high as before.

We have explained how each policy proceeds and provided situations where each policy is more or less performant. We now want to implement each situation to observe the predicted performance. To do so, we use the object oriented language C++ for the simple and the intuitive cache implementation and manipulation.

4. Cache implementation

We know expose the implementation of the cache to get an overview of the methods complexity and be able to analyse the performance result.
We implement three different cache objects: `RANDOMCache`, `LRUCache`, `LRU_MINCache` which extend a parent cache object `ParentCache` that defines common cache attribute and methods. Each cache object has common and more specific attributes. Both methods and attributes may be used either for caching operations or measurement operations. In this part, we will only describe the cache methods specific to caching operations.

```
ParentCache :
Protected :
Cache cache
int sz = 1024*i, i in {50, 60, 70, 80, 90, 100}
Public :
bool isUrl(string url);
void Replace(string body, string url);
string Retrieve(string url);
```

Figure4: `ParentCache` class

The `cache` attribute is the same for all the caches and is a `map<string url, string body>` structure. Since the C++ map has a binary search tree structure, the access costs $O(\log sz)$ with sz the size of the cache.
`sz`: In the rest of the paper, we call cache size the cache memory size. We set it to be a multiple of 1024B with multiplication factors in {50, 60, 70, 80, 90, 100}. In the experiment, we will refer to these factors when we talk about the cache size rather than the real bit size.
`isUrl`: It takes an URL as an argument and return true is the web files corresponding to this URL is in

the cache, false else way.
`Replace`: It takes an URL and the corresponding web page and stores them in the cache according to the specified policy.
`Retrieve`: It takes an URL as an argument and returns the web page corresponding to this URL stored in the cache. It assumes that the web page is already stored in the cache.

We now specify the implementation of each cache and the corresponding complexity operation.

4.1 Random policy

The methods `isUrl` and `Retrieve` use the C++ access methods of the `map` structure to operate so we only specify how the `Replace` method operates.

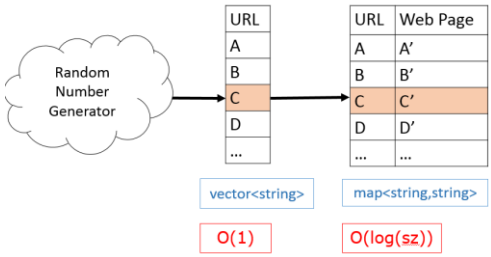


Figure5: RANDOM Replacement Policy

Each time the proxy needs to evict a page, we samples a random integer i between 0 and `cache.size()` and retrieve the i^{th} entry in the cache using the following layer of indirection. A duplicate of the cache index implemented in a `vector`. We chose this implementation for computational complexity. Indeed, we could have accessed the i^{th} element of the map using an iterator and get it to the i^{th} entry by this operation is linear with i while accessing a vector requires constant time. Once again, we face the trade-off between time and space complexity. Even though, it requires an additional storage structure and then more memory space, we choose to favour computational time over memory.

The total complexity of each algorithm:
-retrieval algorithm: $O(\log sz)$
-replacement algorithm: $O(\log sz)$

4.2 LRU strategy

The proxy maintains a double linked-list of index sorted so that the LRU item is the head of the queue. We used the `deque` C++ structure implemented as so. This sorting is maintained through the `retrieve` method. Each time the proxy serves a request thanks to its cache, i.e. when there is a hit for the request of the i^{th}, we move the item to the head in the following way :
-delete the i^{th} item: $O(1)$
-pop it to the head: $O(1)$
-return the page to the client: $O(\log sz)$

Then when the proxy needs to evict a page to make room for a new one, it only needs to
-get the index of the LRU item: $O(1)$
-delete the corresponding entry from the map $O(\log sz)$
-delete it from the index: $O(1)$

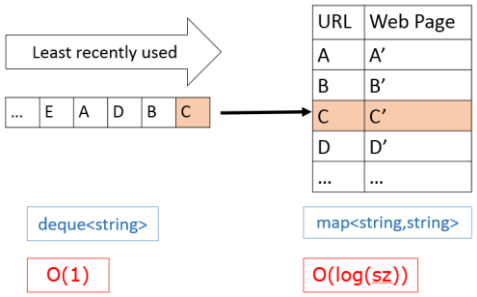


Figure6: LRU Replacement Policy

We sum up the total complexity of each algorithm:
-retrieval algorithm: $O(\log sz)$
-replacement algorithm: $O(\log sz)$

4.3 LRU-MIN strategy

The proxy has the same index mechanism as before and the same `retrieve` method. We now describe the `replace` algorithm.

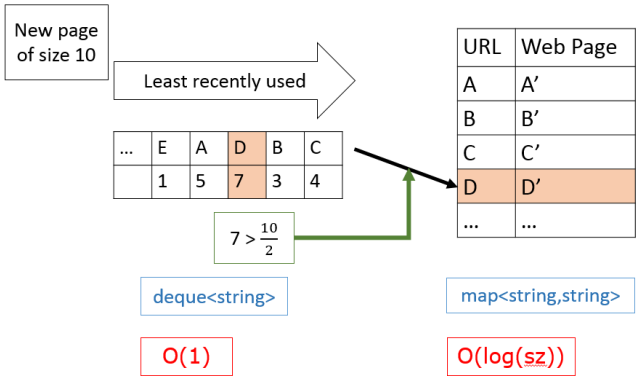


Figure6: LRU-MIN Replacement Policy

Given the size S of the page to put in the cache, we go through the list and:
- check if the size of the entry corresponding to the URL T is bigger than $S_0 = \frac{S}{2}$: $O(\log sz)$
-If so, delete the corresponding entry from the map: $O(\log sz)$
-delete the index from the index list O(1)

We sum up the total complexity of each algorithm:
-retrieval algorithm: $O(\log sz)$
-replacement algorithm: $O(sz * \log sz)$

5. Cache performance

Now that we have chosen caching policies and their implementation, we measure each caching policy performance through two criteria: the hit rate and the service execution time. We define the hit rate by

$$hit\ rate \triangleq \frac{number\ of\ hits}{number\ of\ requests},$$

It measures how smart the cache replacement is: the higher the hit rate, the smarter and the less service latency. We observe the variations of these two measures with respect to time, cache size, the caching policy and the URL workload.

The service time is the time the proxy takes to serve the client request. It is correlated to the hit rate but we still measure some relevant samples of service time to strengthen the hit-rate measurements.

5.1 Hardware and network description

All the experiments are run on two computer lab machine both running on Ubuntu 14. One of the machine will act as a proxy server and the other one as the client and both are connected through a local area network.

5.2 Workloads

As we explained in Section 2, some policies may perform better than other depending on the web requests. We illustrate the "bad" case for LRU and RANDOM when it serves an "outlier" request i.e. a "big page request" while the cache only contains small page. Such a workload can be achieved with a list of URL sorted by increasing page size. We will call this workload `Workload2`. To evaluate the drawback of the previous workload, we build a default workload made of the previous URLs randomly distributed. We will call this workload `Workload1`. We write both of these workload in `url_sort.txt` and `url_shuffle.txt` respectively and used C++ read file operations to make the client request the wanted URLs. The URL were chosen according to their size so that the sorted workload would have more "small" pages than "big" pages but with relatively low variance in the page sizes. The detail of the workload is in the file `url_size.csv` provided with the code. We plot the distribution of the workload page size in Figure

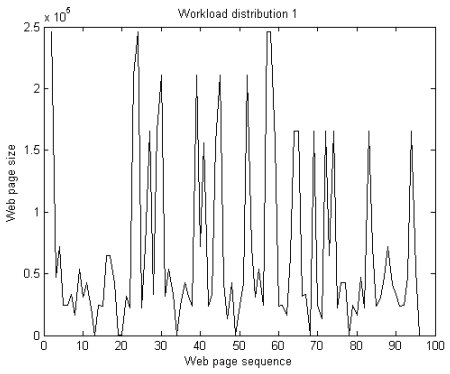


Figure7.a: Uniform distribution of URLs

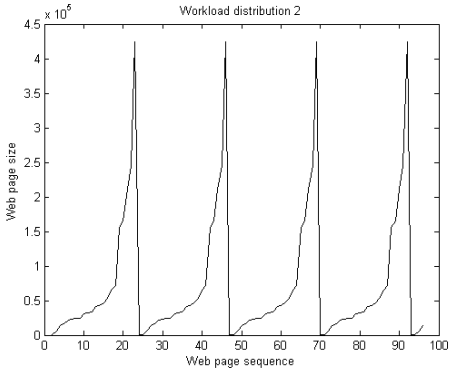


Figure7.b: Sorted distribution of URLs (by page size)

5.3 Hit rate measurements

We measure the evolution of the hit rate with respect to various parameters: time, cache-size, and workload.

5.3.1 Time

For a fixed uniform workload, a fixed cache size, we measure $(n, hr(n))$ the pair of time and the hit-rate at time n. If the caching policy is relevant for the workload, we expect the hit rate to be the highest and "stable". If it the variance of the hit rate is high, it means that the caching policy has not the optimum strategy for the particular workload. Indeed, high variance means that at some time the policy took the optimal decision in the short term but not for a long term optimization as in the case study previously discussed.

$$average\ hit\ rate = \frac{\sum_{time} hit\ rate}{time}, cache\ size\ fixed$$

We use a discrete time, discretized by the request i.e. $t = n$ when the n[th] request occurs. We do not use continuous time for now since our goal is to compare the hit rate between several caching policies when the proxies have undergone the same work. Since we expect some caching policies to perform better than other, if we measured the hit rate with respect to physical time, we would observe a drift of the measurements between the different caching policies. Then at any time t, we would observe the performance of the different models in different environment which is not relevant.

For each policy, we run the experiment by measuring the hit rate after each request and plot its evolution with time/the number of request. Given this, we measure the variance of the hit rate.

$$variance\ hit\ rate = \frac{\sum_{time}(hit\ rate - average(hit\ rate)^2}{time}$$

5.3.2 Cache size

For a fixed uniform workload and a fixed caching policy, we measure the evolution of the average of the hit rate with time for different cache size. We expect the average hit-rate to grow with the cache size: the bigger cache space, the more files the proxy will have in its cache and then the higher the hit rate.

$$hit\ rate\ time\ average = \frac{\sum_{time} hit\ rate}{time}, cache\ size\ fixed$$

We also measure the variance of the hit rate with respect to the time to know if the cache size influences the caching policy relevancy given the same workload.

$$hit\ rate\ time\ variance(cache\ size) = \frac{\sum_{time}(hit\ rate - average(hit\ rate)^2}{time}, cache\ size\ fixed$$

We expect the variance to increase: when the cache size grows, there will be less cache miss so less successive cache miss that i.e. the hit rate will vary less. This can be observed graphically: when the cache size grows, the hit rate variations will decrease and may even converge more or less rapidly.

5.3.3 Workloads

For a fixed cache size, we plot $(n, hr(n))$ for both workloads for each caching policy and observe the graphical result to observe the proxy behaviour. We compute the average of absolute hit-rate difference: the biggest it is, the more sensitive to the work load the caching policy is.

$$average\ hit\ rate\ difference = \frac{\sum_{time}\ \ abs(rate(workload1) - hitrate(workload2))}{|\{cache\ size\}|}, time\ fixed$$

5.4 Service execution time

We now measure the time the proxy server takes to serve the client in different settings. We measure this variable with respect to time, cache size and workload. This is done to correlate the variations of hit rate with the variation of execution time. All the measurements are done in the same conditions as before where $(n, hr(n))$ / $(t, hr(t))$is replaced by $(n, te(n))$ / $(t, hr(t))$ where $te(n)$ is the execution at $t = n$. In fact, we run both measurements of hit rate and execution time with the same measurement. This way, we ensure that these two data can be correlated.

6. Results

6.1 Hit rate

6.1.1 Time

We first observe the $(n, hr(n))$ for the three policy a fixed cache size of 100 on the uniform workload (figure 8). It is the LRU-MIN (red) strategy that gives the best hit rate all along the experience while

LRU(blue) and RANDOM(green) gives approximatively the same performance result. This shows the importance of a long term replacement strategy implemented by LRU-MIN for web calls without identified patterns. That is if the designer must choose a caching policy for a proxy who has not an answer pattern, he should use the LRU-MIN strategy to get the most chance of having a high hit rate. Indeed we measure a higher average hit rate for this strategy than the other as shown by the average hit-rate value.

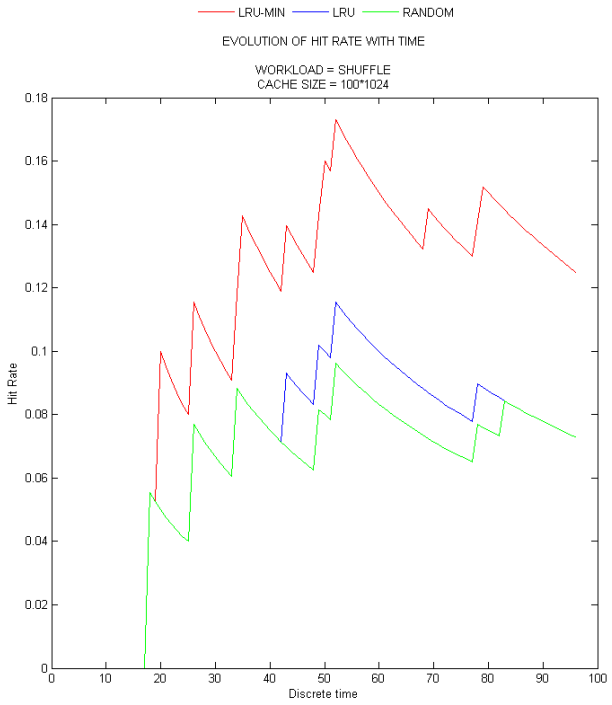


Figure8: Evolution of hit rate with time

However this method does not adapt to the work load the best since it presents the highest variance in the hit rate.

Average hit rate		
LRU-MIN	LRU	RANDOM
0.1084	0.0679	0.0609
Variance of the hit rate		
LRU-MIN	LRU	**RANDOM**
0.030	0.0012	**9.0422e-04**

6.1.2 Cache size

We now observe the evolution of the hit rate with the cache size.

First we compare how each caching policy reacts to changes of the cache size. As expected, the average hit rate grows with the cache size but only until an optimal size which is the same for all the caching policy: 70. After this value, the average hit rate decreases even though it stays close to the maximum value. Given that the only parameter that stayed fixed during this experiment is the workload, we deduce that the optimal cache size is correlated to the workload even though the cache size has different influence on each caching policy. For example the average hit rate for the random policy tends to decrease when the cache size is bigger than 70 while both LRU and LRU-MIN hit rate decrease then increase.

We deduce from this experiment that the cache size must be carefully set according to the workload and the designer must check from times to time if the requests pattern stays the same or not. He can set the optimal cache size through the previously described experiment.

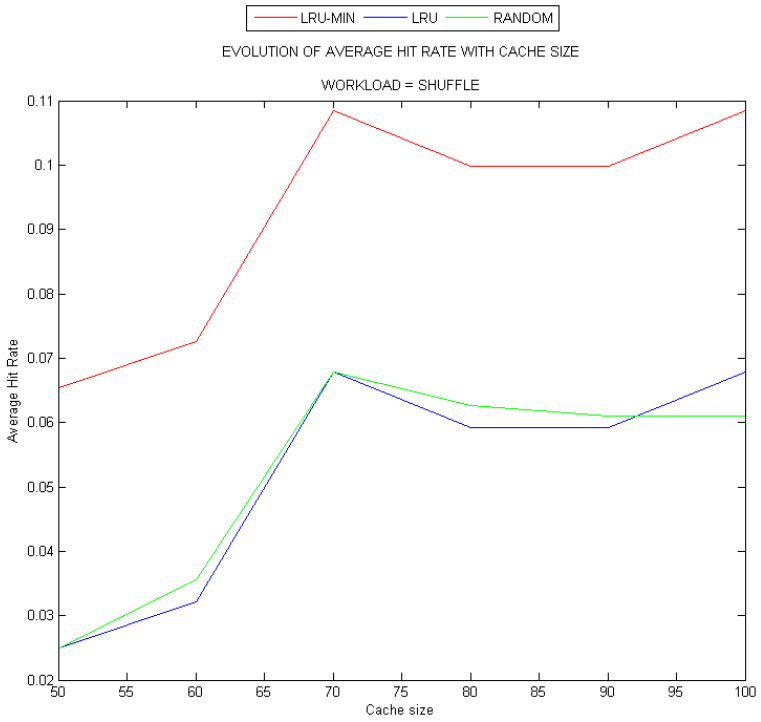


Figure9: Evolution of the averga hit rate with cache size

We now study how each caching policy relevancy changed. First we plot the new hit-rate variations with time for each policy and the different cache size. The following figures plot the variation of the hit rate with time for different cache sizes and for LRU-MIN (left), LRU(right), RANDOM(down) (Figure11). The local time hit rate shows the same tendency as its average that is for a given time, the hit rate increases with the case size inferior to 70 and then have irregular variations. We highlight the curve corresponding to the cache of size 70.

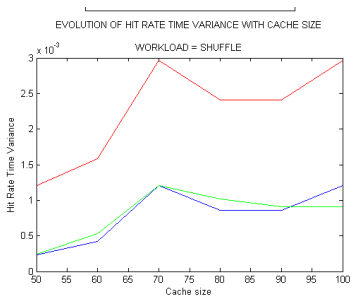
When we measure the variance of the hit rate with respect to the cache size,

$$hit\ rate\ time\ variance(cache\ size) = \frac{\sum_{time}(hit\ rate - average(hit\ rate)^2}{time}, cache\ size\ fixed$$

we find values in the order of 10^{-4} for LRU and RANDOM and an order of 10^{-3} for LRU-MIN. When compared with the order of the hit rate,10^{-2}, this gives us a numerical value of the degree to which we can expect the cache size to influence hit-rate.

Contrary to what was expected, the variance increases with the cache size. Still, it is interesting to notice that the cache size for which this variance is maximal i.e. the cache size that influences the most the hit rate is around 70 for all policies (Figure10). This experiment then tells the designer which is the critical cache size for which the hit rate vary the most i.e. the service latency will vary the most and create random bottlenecks. It is then up to him to tune the cache size to fine the right trade-off between average hit rate i.e. average latency and hit-rate variance i.e. bursts of latency.

Figure10: Evolution of time hit rate variance with cache size
LRU-MIN: red
RANDOM: green
LRU: blue



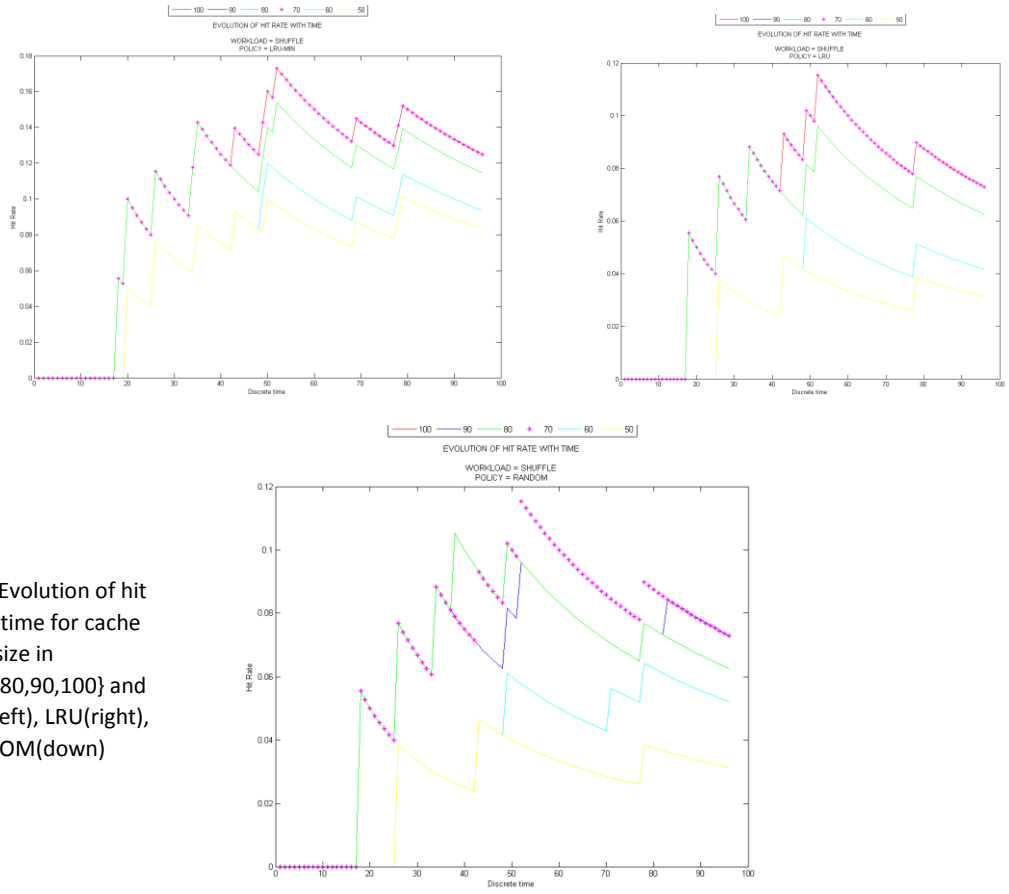


Figure11: Evolution of hit rate with time for cache size in {50,60,70,80,90,100} and LRU-MIN(left), LRU(right), RANDOM(down)

6.1.3 Sensitivity to workload

We start by observing the evolution of the hit rate with respect to time for each policy on the two workloads (Figure 12). As expected in the theoretical study in Section 2, the LRU and RANDOM strategy never get hits with the sorted workload while the LRU-MIN strategy gets to maintain its hit rate. Once again this shows the importance for the policy to take into account long-term decisions especially when the proxy serves such a regular workload as this one. Then the cache designer is invited to study first the request pattern of the clients: a well define model will help him choose the right policy and increase drastically his cache performance. This also shows the importance of long term strategy since the sorted distribution induced a 30% decrease of the average hit rate compared to 100% for the other policies.

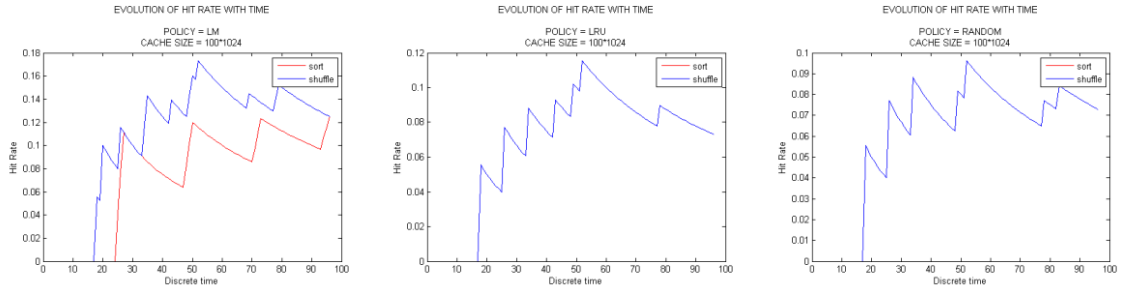
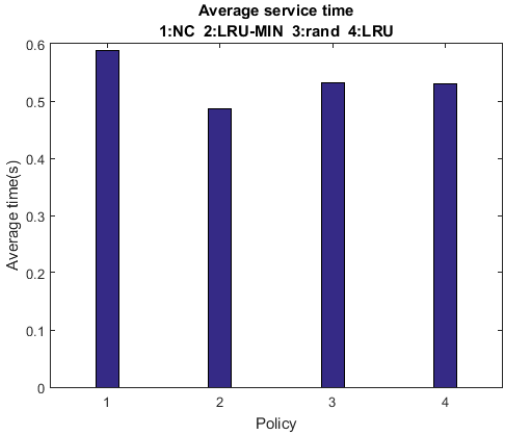


Figure12: Evolution of the hit rate with time(LRU-MIN, LRU-RANDOM)
Sorted workload : red
Uniform workload : blue

6.2 Service Time

Now that we know how to tune the cache-proxy server to get optimal result (i.e. set a cache size of 70), we compare the service time of the four following proxy server: without cache(NC), with the LRU-MIN， RANDOM, LRU caching policy.



As expected, we measure that all the cache proxy perform better than a proxy without cache which role is to relay the client's request. Regarding the caching policy, we find the expected correlation between hit rate performance and service time performance: the LRU-MIN which performance is 17% higher than the NC strategy. As for the RANDOM and LRU strategy, the also show an improvement of the performance of respectively 9% and 10% with respect to NC.

7. Conclusion

We gained several insights regarding the implementation of web server. First, it is relevant in term of time service and server contention to use proxy server to distribute client's requests among them. These proxy servers prove to be all the more performant when they have storage memory and implement the relevant caching policy.
The caching policy performance measured by the hit rate proved to have a high correlation with the time serve. The optimum policy depends on both client request pattern and the cache memory size. Still, we also observed that the optimum cache size depends on the request pattern. So we can deduce that the optimum policy depends only on the client pattern from which we deduce the right cache size to adopt. This shows the importance to analyse the client request traffic to first build a statistical model of their request and then get the optimum caching policy and then optimize the proxy server performance. It is also important to recall that this model may change with time so the model must be updated regularly. Without such model, the most performant and robust strategy to cache size and workload variations proved to be the LRU-MIN strategy that favours long term benefit (high average hit rate) over short term benefit (relatively costly replacement algorithm).