Architectures and Performances: First project

ISTASSE Maxime 5679-12-00
JACQUET Charles 2781-12-00

October 2015

Introduction

In this project, we have been asked to implement popular cache management strategies and to evaluate their performances against a realistic dataset. As this dataset is from another time, we decided not to allow too much space to it, and limited ourselves to 64MB of cache. Because the average file is about 21152 bytes, we took a maximal N of 3000 for both the first task and the second, which we'll see, can already seem pretty big.

1 Warm-up phase

Since our cache has a maximal size of 3000 average files, and we would like to have the majority of the space occupied in the cache, having a X bigger than 3000 seems legitimate. 3000 would be perfect if we had distinct files requested. But as the hit ratio is about 80%, this means that a good part of the requests were already duplicates. We have in fact loaded about $0.20 * 3000 = 600$ unique files at maximum. In order to fill around 75% of its capacity, we would plan on $X = 20000$, with about $20000 * 0.11 = 2200$ files loaded in the warm-up phase and still not waste much time. The numbers we are talking about are illustrated on figure 1. We can see that both LRU and LFU have the same hitrate, and this is obvious, because none of them is full yet.

In reality, half of that space is occupied in the cache, because of the size conflicts for a same URL. But we still thought this was a good number to base our tests.

2 Task 1.1

For now, let's consider we have files taking one slot in the cache, and our cache has a fixed number of slots.

2.1 LFU implementation

We have implemented the LFU strategy using a standard Java HashMap to perform the lookup fast from the URL of the request, and a PriorityQueue to handle the removing of the least frequently used element. Both make reference to a ResNode object, which consists in a Resource (URL, size), a frequency counter, and a timestamp to differentiate the newly added and remove the oldest in case several files have the minimal frequency.

In case we have to remove the least frequently used element, we just have to poll it from the queue ($\Theta(1)$) and remove it from the map. (around $\Theta(1)$) In case of hit, we just have to remove it from the queue, increase the counter, and put it back. This is $O(log(size))$. On first caching, we just have to put the entry in the map and in the priority queue. ($\Theta(1)$ and $O(log(size))$)

Of course, we don't use the timestamp to optimize our strategy, this is just the way we found to make the priority queue act as we wanted, we are aware there should be more efficient ways to handle that for real life applications.

2.2 Hitrate vs cache size

We have obtained the hitrates for cache sizes from 1 to 3001 showed in figure 2, considering the whole trace. We have kept the warm-up parameter of $X = 20000$ requests. We can here see that $N = 3000$ is already quite a good parameter. In fact, we have looked a bit further, and having a bigger cache didn't improve the hitrate significantly.

3 Task 1.2

In this section, we'll keep the warm-up parameter $X = 20000$, but we are going to consider the cache as a 64MB one, taking the size of every file inside it into account, but not the memory segmentation.

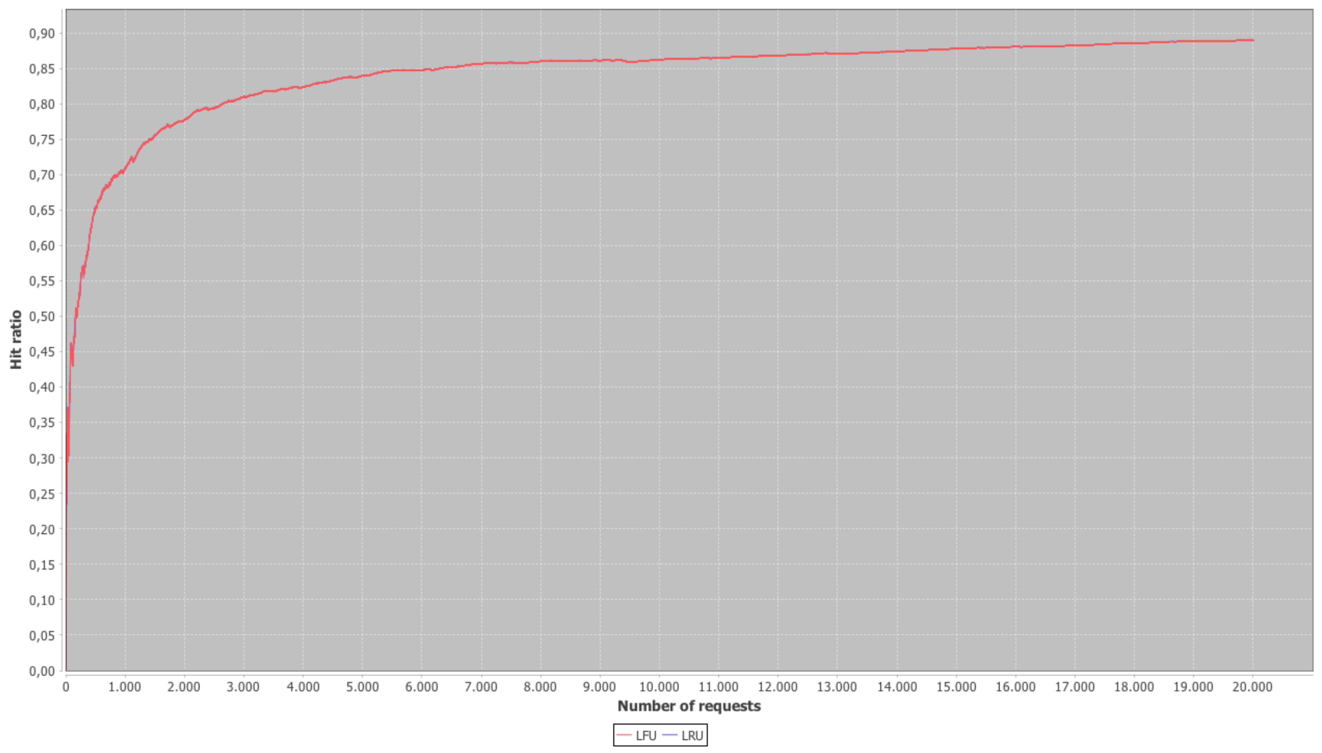

Figure 1: Evolution of the hitrate on the 20 000 first requests, without warm-up

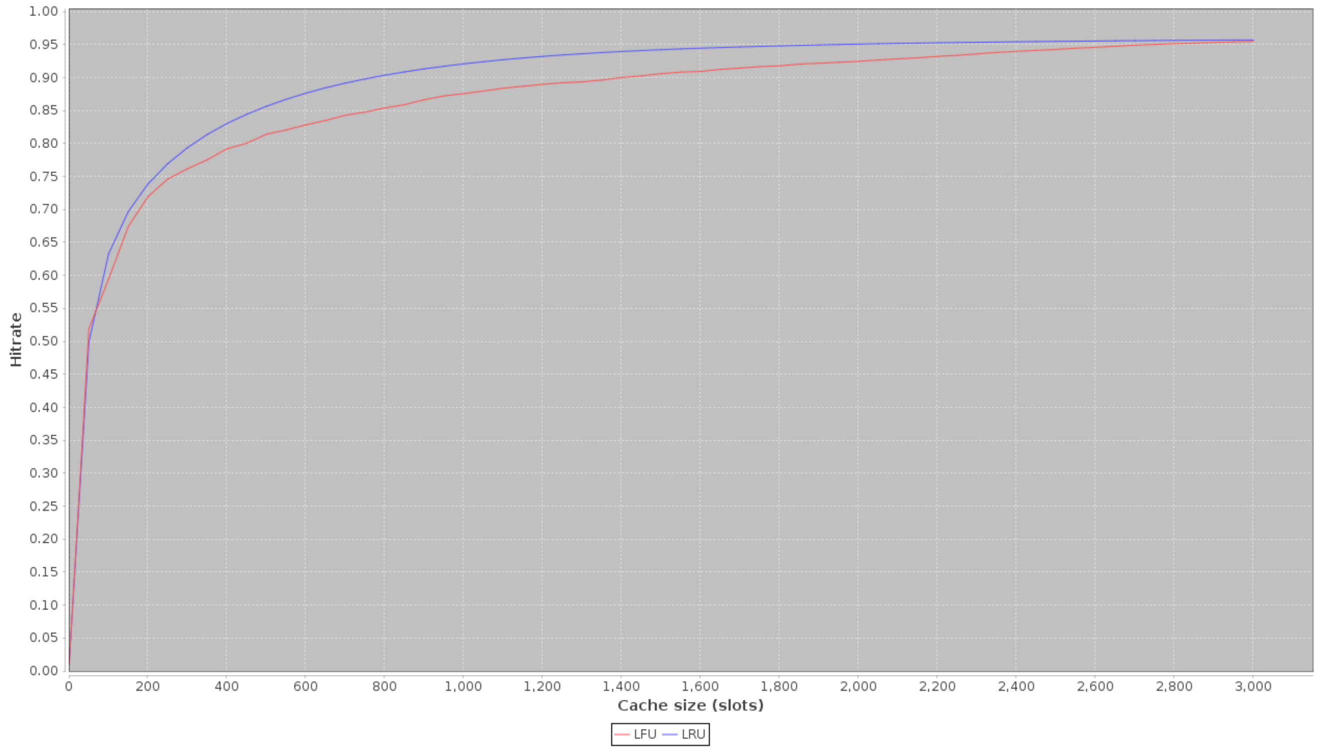

Figure 2: Evolution of the hitrate with respect to the cache size in slots (LRU and LFU)

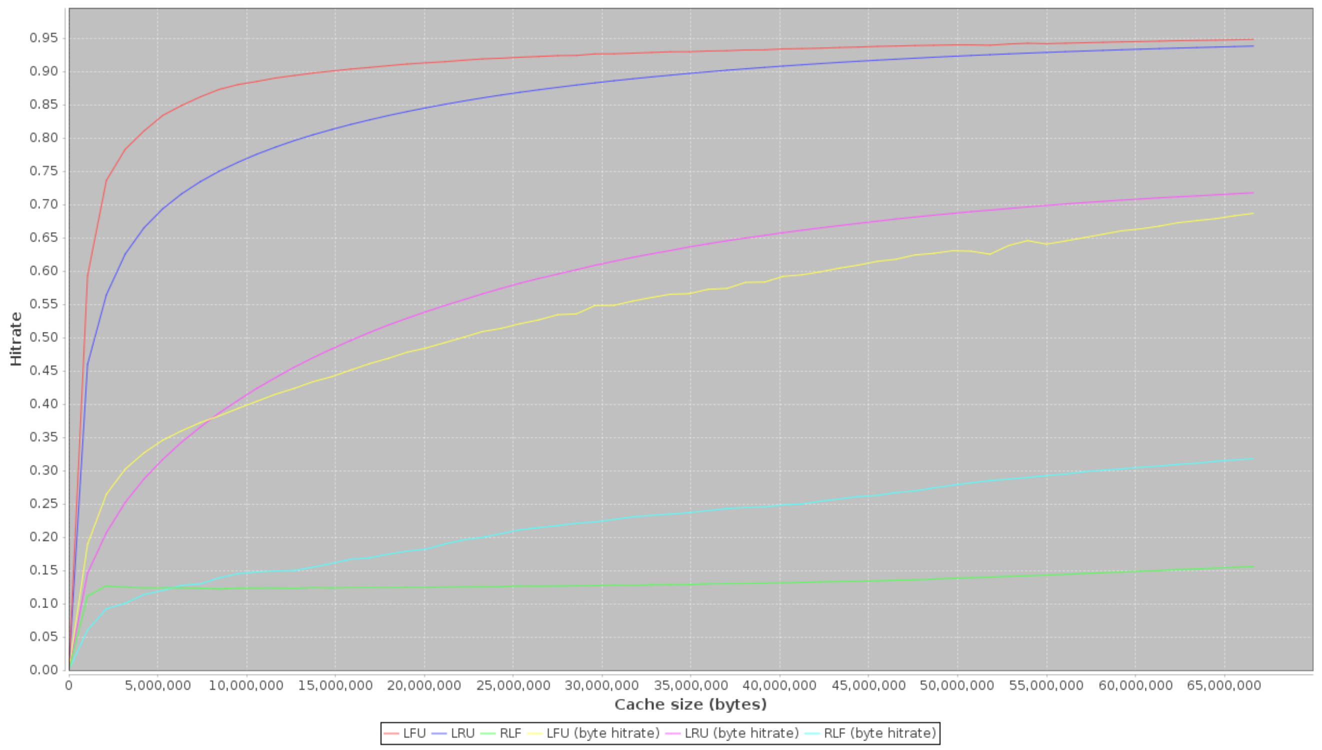

Figure 3: Evolution of the hitrate with respect to the cache size in bytes (LRU, LFU and Size-based (Remove Large First))

3.1 Remove Large First (RLF) strategy

The implementation is the same as for the LFU strategy except for the comparison implementation, so still using a PriorityQueue to remove largest files first, and a HashMap for the lookup. We have chosen to remove largest resources in order to keep more elements into the cache and improving the hit rates for small elements.

3.2 Performance discussion

On Fig.3 we see that best byte rates are for the LRU and LFU which ones are really close to each other. And we clearly see that our remove large first strategy is not as effective as the others. It can be explained by the fact that larger files are actually more frequent and moreover, it's easier and more interesting to charge again small files unlike big ones.

Conclusion

We have seen in this report that no caching strategy wins over all the others. A lot factors impact the performance for real life applications, like the size of the objects we have to cache, the size of the cache, the behavior of the service accessing the cache, ... We have the feeling of having been quick about some topics, but it wasn't requested, and since we already have a bit more than 2.5 pages with the plots, we didn't put it in our report.