Improving performance of multiple-level cache systems

Ismael Rodríguez
Universidad ORT Uruguay
rodriguez_i@ort.edu.uy

Andrés Ferragut
Universidad ORT Uruguay
ferragut@ort.edu.uy

Fernando Paganini
Universidad ORT Uruguay
paganini@ort.edu.uy *

ABSTRACT

Cache networks architectures have been widely deployed as a response to the growth in content demand. This topic has been widely studied through theoretical and empirical approaches. In this paper we evaluate different caching policies for an array of caches, and show that performance can be improved by incorporating the idea of pushing recently evicted content upstream. Both numerical and analytical studies are provided, and these show promising results for future lines of work.

1. INTRODUCTION

The explosive demand for certain files in the Internet has led naturally to the deployment of cache systems, which improve responsiveness by storing content closer to clients. The main design issue is how to manage which file to store, and which to evict, under limited storage capacity.

Most of the strategies deployed for this purpose have been conceived from the point of view of a *single* cache. In particular, a popular algorithm is to cache the most recent request and evict the Least Recently Used (LRU) file. While the exact analysis of this policy is difficult, approximations [3] can be used to support its good performance. An alternative approach, more amenable to theoretical studies are Time To Live (TTL) caches, where evictions are determined by a timer associated with each stored content.

When *networks* of caches are considered, it is often assumed that each cache makes decisions independently, and the collective performance must then be assessed. Here again the LRU policy is harder to analyze, with more exact results available for the TTL case [2, 5]. However, performance gains could be obtained from algorithms that couple decisions at different caches; a series of such alternatives are surveyed in [6].

In this paper we add to these choices the idea of not discarding evicted files, but rather "pushing" them upstream in the cache hierarchy, since recently evicted files may still be in demand. This leads us to the "Move to Front - Push" and the "Pull-Push" algorithms, which we describe in Section 2, and compare in Section 3 by simulation experiments using LRU caches. In Section 4 we assume instead that caches follow a TTL policy, and analyze their performance using Markov chain methods. Conclusions are given in Section 5.

2. MODEL AND ALGORITHMS

A cache network is generally defined (see e.g. [9]) by a set of cache/routers with the capacity of storing files and also forwarding requests. The network has at least one *custodian* which stores all files. File requests may be received at different nodes; if the file is available locally we have a *hit*, otherwise a *miss* occurs and the request is forwarded to another cache. The topology must guarantee that, under successive misses, requests eventually reach the custodian.

Once a hit occurs, the file is returned to the client along the exact same path through which requests were forwarded. This fact permits the implementation of different inter-cache algorithms; we list here the main ones from [6]:

- **Leave Copy Everywhere (LCE):** A copy of the file is stored in each of the caches of the path.
- **Move To Front (MTF):** The content is only stored in the cache where the client requested it.
- **Move Copy Down (MCD) or "Pull":** In this case the file is moved one cache closer to the client, instead of moving it all the way down.

In all the above alternatives, it is generally assumed that if any cache capacity is exceeded, the evicted file is discarded. In this paper we explore a simple alternative: instead of discarding, **push** the evicted file to the next

*The authors were partially supported by AFOSR US under grant FA_9550_15_1_0183.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LANCOMM, August 22-26, 2016, Florianopolis , Brazil
© 2016 ACM. ISBN 978–1–4503–4426–5/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2940116.2940119

cache up the forwarding path[1]. The idea is that if the file was cached until a moment ago, it is likely to still be useful to local clients. Note that this procedure would have limited impact in the LCE policy, since in that case the file is likely to already be stored upstream; in the other cases it is a more significant variation. Hence, we define the following two policy variants:

- **MTF-Push:** Adds the push feature to MTF.
- **Pull-Push:** Adds the push feature to MCD (Pull).

In what follows we will analyze the performance of these policies, in comparison with the standard ones.

For simplicity we will focus here on the case of a line topology, with N caches. Cache k has storage capacity B_k; this will be considered a hard bound in the case of LRU policies, and a mean occupation bound for studies with TTL caches. Client requests are assumed to arrive at cache 0, and the custodian is located after cache N. More general studies of these policies over other topologies were carried out in [8].

3. EXPERIMENTS WITH PUSH/LRU

We carry out numerical comparisons by simulating the different algorithms, using as metrics: (i) the mean number of hops until the content is found (which is a measure of delay in content retrieval); and (ii) the fraction of time each content file is stored in each cache.

As a simple example to test performance, we use a line topology composed by four buffers with capacity for $B = 10$ files; there is a total of $N = 100$ files assumed of equal size, with a popularity distribution following Zipf's Law: ($p_m = \frac{1}{m^\alpha}$); this is commonly accepted [1] for Web documents. For simplicity we assume requests arrive according to a Poisson process while evictions follow the LRU rule. In Table 1 we compare the mean delay of each algorithm, varying the Zipf parameter α.

α	LCE = MTF	MCD	MTF - Push	Pull - Push
0.0	7.176	6.269	5.998	6.001
0.8	5.773	4.495	4.213	3.564
1.2	3.449	2.733	2.267	1.878
2.0	0.541	0.563	0.313	0.296

Table 1: Mean number of hops to retrieval.

Note that with requests arriving at a single node, MTF performs equally to LCE: the first cache has identical behavior, and additional caches are not useful (in MTF they are empty and in LCE they have the same contents as the first). The remaining three policies differentiate the content in the upstream caches: this is one reason for their improved performance.

More importantly, we see a clear improvement due to the **push** feature, across values of α. This validates the heuristic of keeping recently evicted content close.

[1]Of course, this may trigger further evictions upstream.

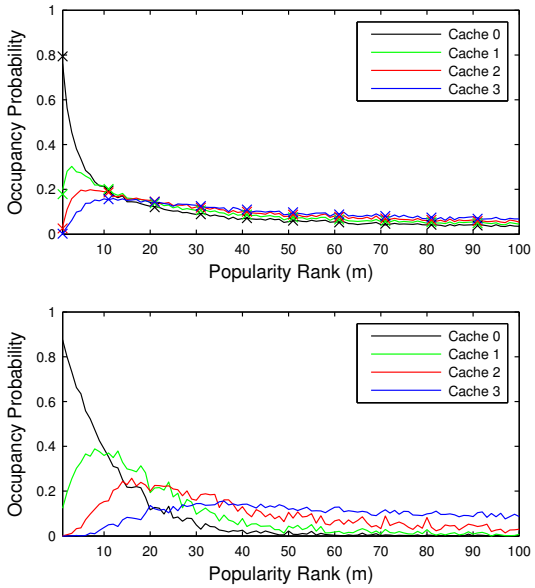

Figure 1: Storage probabilities per file and cache. Top: MTF-Push, '×' indicates Che's generalization. Bottom: Pull-Push.

In fact for a hypothetical central planner who knows file popularities, under exponential inter-request times, the optimal storage policy is to *statically* store files with distance to the clients a decreasing function of popularity: namely, store the B_0 most popular files in the first cache, the next B_1 in the second cache, and so on. This property has been known since [7] for a single cache under the so-called Independent Reference Model for requests, which holds for Poisson arrivals; the extension to a line of caches is straightforward. We note that this optimality may *not* hold for general inter-request times, as established in [4].

Does the **push** heuristic achieve (approximately) the optimal file distribution? To understand this we turn to the graphs in Figure 1. The horizontal axis indexes files in decreasing popularity; the vertical axis is the fraction of time the file is stored. The different caches in the line are represented, separately for the MTF-Push and Pull-Push policies.

The MTF-Push system achieves only a modest differentiation between the occupancy distribution of the different caches, noticeable mainly in the most popular files. In this case we also validated the results with an analytical model: Che's approximation to LRU [3]. The model can be applied to the first LRU cache on its own, but also under MTF-Push the first two caches behave collectively as an LRU cache, and so on, from where individual occupancies are derived. We observe a close match with experimental results.

On the other hand, the Pull-Push algorithm achieves a closer to optimal distribution: the first cache stores mainly the most popular contents, while the second

spreads the occupancy among the next most popular contents. This is consistent with the better performance of the Pull-Push algorithm; analytical models are not available for this case.

4. ANALYSIS OF PUSH FOR TTL

We now turn our attention to TTL caches. Here, when caching a file a timer is started, and the file is evicted upon its expiration; if another request is received before that, the timer is reset. Assuming evictions only occur due to the timer, the occupation process for each file can be analyzed in a decoupled way. This premise implies that buffer bounds are not hard, but instead can be imposed in an average sense, which for large enough systems is well justified [4].

Suppose now that we have a line of caches as considered before, under the **push** strategy: evicted files from one TTL cache are pushed upstream; both the Pull-Push or MTF-Push alternatives will be considered. For simplicity, assume the line of caches is infinite in length: this means there is exactly one copy of each file m in the cache system, whose location evolves over time driven by the request and timer processes. If we can calculate the probability (proportion of time) $\pi_k(m)$ that file m is stored in cache k, then the mean number of hops to find the file is[2] $c_m = \sum_{k=1}^{\infty} k.\pi_k(m)$.

We now calculate $\pi_k(m)$ for both strategies assuming timers are *exponentially* distributed with mean $1/\mu_m$, which may be file dependent. The evolution of the file location in the line of caches is then given by a continuous time Markov chain, shown for each case in Figure 2. Analyzing them we find that the steady-state distribution is in both cases *geometric*, of the form

$$\pi_k(m) = (1-\rho_m)\,(\rho_m)^k\,.$$

The only difference is in the expression for ρ_m, namely

$$\rho_m^{P-P} = \frac{\mu_m}{\lambda_m}; \qquad \rho_m^{M-P} = \frac{\lambda_m}{\lambda_m+\mu_m}.$$

We should note as well that the Pull-Push case has the stability condition $\mu_m < \lambda_m$ (the timer must be on average slower that the inter-request rate), whereas MTF-Push is always stable. The resulting expression for the mean number of hops to reach the file is

$$c_m = \sum_{k=1}^{\infty} k.\pi_k(m) = \frac{\rho_m}{1-\rho_m}.$$

Suppose one could choose the timer parameter μ_m for each m; a natural optimization would be

$$\min \sum_m \lambda_m.c_m \quad s.t.: \sum_m \pi_k(m) \le B_k \;\text{ for each } k.$$

As both policies' occupancy probabilities are decreasing in k, if all buffers have the same size B_0 we may impose the above restriction only at $k=0$. The main observation is that for both policies above (indeed for any policy that yields a geometric distribution) we have $\pi_0(m) = \frac{1}{1+c_m}$; therefore our optimization can be restated as the convex program:

$$\min \sum_m \lambda_m.c_m \quad s.t.: \sum_m \frac{1}{1+c_m} \le B_0, \qquad (1)$$

in particular having the *same* optimum. So both policies involving **push** can achieve the same performance under TTL if the μ_m's are optimized accordingly.

[2]This assumes that requests find the system in a typical state; this is the PASTA property of Poisson arrivals.

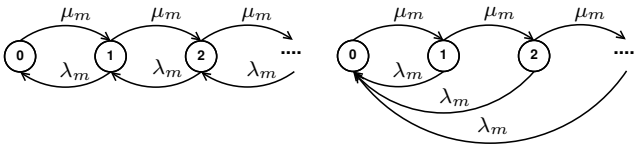

Figure 2: Pull-Push (left) and MTF-Push (right) associated Markov Chains

5. CONCLUSIONS

In this paper we studied multiple-level cache systems, focusing on a line topology. Under LRU evictions, we showed empirically that **pushing** content upstream improves performance, in two versions (MTF-Push and Pull-Push). We also showed analytically that for TTL caches with exponential timers, the same performance is achievable by both.

Since the optimization in (1) requires tailoring timers to file demands, the equivalence will not hold for LRU, where eviction times are approximately equal across files [3]. Our experiments suggest a preference for Pull-Push, but the question remains open for future research.

6. REFERENCES

[1] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, 1999.

[2] D. S. Berger, P. Gland, S. Singla, and F. Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.

[3] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE J. on Selected Areas in Comm.*, 20(7):1305–1314, 2002.

[4] A. Ferragut, I. Rodriguez, and F. Paganini. Optimizing TTL caches under heavy-tailed demands. In *ACM/Sigmetrics*, 2016.

[5] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

[6] N. Laoutaris, S. Syntila, and I. Stavrakakis. Meta algorithms for hierarchical web caches. In *IEEE ICPCC*, pages 445–452, 2004.

[7] Z. Liu, P. Nain, N. Niclausse, and D. Towsley. Static caching of web servers. In *Photonics West'98 Electronic Imaging*, pages 179–190, 1997.

[8] I. Rodríguez. Diseminacion de contenidos en redes de datos distribuidas. Eng. Thesis, Universidad ORT Uruguay, 2015.

[9] E. J. Rosensweig, D. S. Menasche, and J. Kurose. On the steady-state of cache networks. In *IEEE/Infocom*, pages 863–871, 2013.