# Expected competitive ratio for evaluating scheduling algorithms of multi-user cache

Extended Abstract

Daniel Berend[1], Shlomi Dolev[1], Avinatan Hassidim[2], Marina Kogan-Sadetsky[1]

**ABSTRACT**

We present an efficient realistic metric for evaluating cache scheduling algorithms in multi-user multi-cache environments. In a previous work, the requests sequence was set deliberately by an opponent (offline optimal) algorithm in an extremely unrealistic way, leading to an unlimited competitive ratio and to extremely unreasonable and unrealistic cache management strategies. In this paper, we propose to analyze the performance of cache management in a typical scenario, i.e., we consider all possible scenarios, with their (realistic) distribution. In addition, we present an efficient, according to our novel average case analysis, online heuristic algorithm for cache scheduling. The algorithm is based on machine-learning concepts, and is flexible and easy to implement.

**Introduction and previous work.** Finding an efficient management for multi-user concurrent cache systems is of high interest and importance. A cache scheduling algorithm decides, upon each request that results in a page fault, which page to evict from the cache to insert the newly requested page. The algorithm should minimize the number of page faults.

Offline paging strategies may align the demand periods of future requests. Previous works show that well-known high performance cache management policies for a single user cache management, such as LRU, FIFO, CLOCK, and FWF, are non-competitive in the multi-user setup. They do this by defining an extremely unrealistic request sequence chosen by an adversary, which leads to unreasonable offline cache management strategies and an unbounded competitive ratio. Thus, apparently, one cannot evaluate the real efficiency of the online algorithm. To the best of our knowledge, up to now, there is no realistic basis to measure the quality of online multi-user concurrent multi-cache management algorithms.

**Our contribution**. In this paper, we propose three versions of near-optimal offline algorithms, each with its level of computational complexity, from hard to easy. Of course, as the algorithm becomes simpler, its accuracy decreases. We claim that, even the simplest of the threesome, is a good comparison point with online algorithms.

**System model**. Suppose, in general, that we have servers $S_1, S_2, \dots, S_n$, and corresponding lists of possible requests $R_1, R_2, \dots, R_n$, where $R_i = \{r_1^i, \dots, r_{l_i}^i\}$ is of length $l_i$. The system contains a cache of size $K$, and a slow memory with access latency $t$. We implicitly assume that $K < |\bigcup_{i=1}^n R_i|$. A *singleton* is a subset of size $K$ of $\bigcup_{i=1}^n R_i$, namely, any of the possible cache configurations. Each of the singleton configurations starts an execution tree.

When expanding a node consisting of several singletons, we have branches corresponding to each of these singletons, with possible request configurations. For each request sequence, we leave only those branches yielding a minimal number of page faults, and discard of all others. Suppose we expand the tree from a singleton $A$. Denote by $A_i, 0 \le i < \infty$, the subtree consisting of all nodes up to level $i$ of the full execution tree. We introduce the following notations:

$\boldsymbol{E_A(i)}$ – number of execution paths in $A_i$.

$\boldsymbol{F_A(i)}$ – number of page faults in $A_i$, i.e., the sum of the numbers of page faults over all execution paths in the tree.

$\boldsymbol{P_A(i)}$ – average number of page faults per execution path in $A_i$.

$\boldsymbol{R_A(i)}$ – average number of page faults per request in $A_i$.

We are interested in the asymptotics of $\boldsymbol{R_A(i)}$ as $i \to \infty$. The average of these asymptotic values over all singletons is the baseline (offline bound) we use to calculate the competitive ratio of online algorithms. We construct a system of recurrence equations for these quantities. Once $E_A(i)$ and $F_A(i)$ have been calculated, we readily find $P_A(i)$ and $R_A(i)$: $P_A(i) = \frac{F_A(i)}{E_A(i)}$, $R_A(i) = \frac{P_A(i)}{i}$.

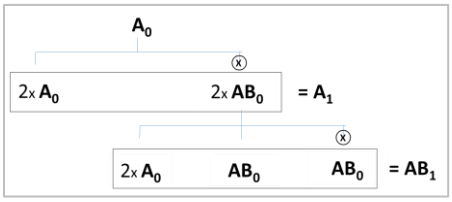


Figure 1: Example of the full expansion of an execution tree, with $K = 3$, and two servers $S_1, S_2$ with corresponding sets of possible requests $R_1 = \{r_1^1, r_2^1\}, R_2 = \{r_1^2, r_2^2\}$

**Heuristic online algorithm**. Given the above, we conclude that learning the recent request probabilities may lead to an efficient cache management that can be compared with the obtained offline bound. We propose an algorithm whose steps include learning the previous time window $\Delta T$ and deriving heavy hitters. As long as the request probabilities stay put, we leave the cache unchanged. The algorithm also detects an optimal execution window size, which is based on the rate of changes.

[1] Ben Gurion University, Israel, (berend/dolev/sadetsky)@cs.bgu.ac.il
[2] Bar Ilan University, Israel, avinatan@macs.biu.ac.il