Average-Case Competitive Ratio of Scheduling Algorithms of Multi-User Cache

Dani Berend
Ben-Gurion University of the Negev
Shlomi Dolev (shlomidoev@gmail.com)
Ben-Gurion University of the Negev
Avinatan Hassidim
Bar-Ilan University
Marina Kogan-Sadetsky
Ben-Gurion University of the Negev

Research Article

Keywords:

Posted Date: April 20th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1562036/v1

Average-Case Competitive Ratio of Scheduling Algorithms of Multi-User Cache

D. Berend[1,2] S. Dolev[2] A. Hassidim[3] M. Kogan-Sadetsky[2]

[1] Department of Math, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel.
[2] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel.
{berend, dolev, sadetsky}@cs.bgu.ac.il.
[3] Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
avinatan@macs.biu.ac.il.

Abstract. The goal of this paper is to 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 possibilities with their (realistic) distribution. In other words, we analyze the average case and not the worst case of scheduling scenarios. 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, it is flexible and easy to implement.

1 Introduction

A multi-user concurrent multi-cache system should satisfy users' memory page requests. Finding an efficient cache management for these systems is of high interest and importance. A cache scheduling algorithm decides, upon each request that results in a page fault (when the cache is full), which page to evict from the cache in order to insert the newly requested page. The algorithm should minimize the number of page faults.

Offline paging strategies may align the demand periods of (known to the offline) future requests. An online strategy does not know the exact sequence of future requests. Thus, an optimal offline strategy has a significant advantage over an online strategy. The sole purpose of considering the offline case is to form a basis for comparison with the online scheduling. For example, if the competitive ratio is 1, then the online algorithm is obviously optimal, as it is not affected adversely by the uncertainty of future requests. Of course, one cannot hope for such a ratio for an online algorithm. Previous works show that, in multi-core

paging, the competitive ratio of traditional algorithms in the scope of multi-cache systems, such as LRU, FIFO, CLOCK, and FWF, may be arbitrarily large.

Both [5] and [7] show that traditional online paging algorithms are non-competitive in a multi-core model. They prove that well-known high-performance cache management policies for a single user cache management, such as LRU, are non-competitive in the multi-core 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 compare the real efficiency of (the offline and) the online algorithm. In particular, to the best of our knowledge, up to now there is no realistic standard to measure the quality of online multi-user concurrent multi-cache management algorithms.

An optimal algorithm described in [7] is to simulate an execution of these sequences, with all the possible ways of serving them, and finally to select the best one, i.e., the one with the smallest number of page faults (#pf). Although this algorithm provides the optimal answer, it is inefficient, and for large systems with long request sequences it is inapplicable.

In this paper, we propose two alternative 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 these three algorithms is a good comparison point versus online algorithms.

2 Related work

Cache performance has been extensively studied in multi-core architectures but less so in multi-server architectures. Some proposed techniques are able to schedule requests [5] so that the paging strategy can choose to serve requests of some sequences and delay others. Thus, the order in which requests are served is algorithm dependent. Lopez-Ortiz and Salinger [7] present another technique that discards this possibility. Given a request, the algorithm must serve the request without delay. The algorithm presented in [7] is limited by a two-level non-distributed memory system. Awerbuch et al. [1] present a distributed paging algorithm for more general networks. Although the algorithm in [1] is applicable to distributed systems, it serves requests in a sequential manner: at any time a processor p may invoke a single request. In our model, requests from all the servers in the system are served simultaneously.

Competitive analysis is a method invented for analyzing online algorithms, in which the performance of an online algorithm is compared to the performance of an optimal offline algorithm that can view the sequence of requests in advance. An algorithm is competitive if its competitive ratio – the ratio between its performance and the offline algorithm's performance is bounded. The quality of an online algorithm on each input sequence is measured by comparing its performance to that of an optimal offline algorithm, which is, for an online problem,

an unrealizable algorithm that has full knowledge of the future. Competitive analysis thus falls within the framework of worst-case complexity [2].

A caching system can use heuristics to determine dynamic partition of the cache [3, 8, 10–12, 6]. The cache scheduling heuristics in previous works are based on dividing the cache between cores, and handling each part independently. The author of [5] shows that, given the amount of cache dedicated to each core, it is easy to decide which location to evict – FITF is the optimal strategy. It mentions that more relevant to practical cases are heuristic algorithms phrased in terms of allocating cache to each core, while in its own allocated share each core should perform LRU.

3 System model settings

Suppose, in general, that we have servers $S_1, S_2, ..., S_n$, and corresponding lists of possible requests $R_1, R_2, ..., R_n$, where $R_i = \{r_1^i, ..., r_{l_i}^i\}$ is of length l_i. The system contains a cache of size K, and a slow memory with access latency time of t. Denote by $R = \cup_{i=1}^n R_i$ the set of all possible requests. We implicitly assume that $K < |R|$, but in practice K is usually much smaller even than each of the $|R_i|$'s separately.

A *singleton* is a subset of size K of R, namely, any of the possible cache configurations. (The term derives from the fact that, in the process of constructing the full execution tree, we often place several possible cache configurations at a node, with the intention of resolving only later the identity of the actual configuration residing there.) 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. In fact, we cross out some of these branches. Namely, each of the request sequences yields some number, between 0 and n, of page faults for each of the singletons. For each request sequence, we leave only those branches yielding a minimal number of page faults and discard of all others.

For each request sequence, and each singleton with a minimal number $f_{\min}$ of page faults in the current node, we need to consider those singletons we may move to. For each singleton, these are the singletons obtained from it by:

i Leaving intact all the requests that are currently in the cache and have been queried now.
ii Replacing any $f_{\min}$ requests currently in the cache, that have not been queried now, by the $f_{\min}$ missing requests.
iii Leaving the rest of the requests untouched.

Now we explain how, given a node and a request configuration, we construct the corresponding child of the node. We go over all singletons in the node. For each singleton, we find as above the set of singletons we may move to. The child consists of the union of all these sets of configurations. A node containing one singleton A is labeled by A, a node containing A and B – by AB, and so forth.

In principle, we need to expand the execution tree so as to deal with all possible sequences of request configurations. However, there is no point in expanding a node that has been encountered already. Hence, we expand the tree only until no more nodes with new content can be generated. We mention in passing that, even if the request sequence is infinite, this allows us to deal with a finite expansion of the full execution tree. In fact, the number of singletons is $\binom{|R|}{K}$. Hence the number of distinct node labels we may encounter is at most $2^{\binom{|R|}{K}}$. Hence, we will never have to expand the execution tree beyond depth $2^{\binom{|R|}{K}}$. This bound is probably way above the actual depth we will get, but serves to demonstrate that the process terminates. When we expand all the nodes, usually some potential node labels will not appear. Moreover, there may well be pairs of equivalent labels, namely, nodes having the same behavior when expanded. When several labels are found to be equivalent, we may replace any occurrence of one of them by that of a representative of the group. This tends to shorten the algorithm a great deal. Our goal is to calculate the expected number of page faults per group of simultaneously arriving requests. As we consider all possible request sequences, so that the full execution tree is infinite, we need to define what this expected number means. Suppose we expand the tree from a singleton A. Denote by $A_i, 0 \leq i < \infty$, the subtree of the full execution tree consisting of all nodes up to level i. (Thus, for example, $A_0 = A$.) We introduce the following notations:
$E_A(i)$ – number of execution paths in A_i.
$F_A(i)$ – number of page faults in A_i, i.e., the sum of the numbers of page faults over all execution paths in the tree.
$P_A(i)$ – average number of page faults per execution path in A_i.
$R_A(i)$ – average number of page faults per request in A_i.

It will be convenient in our calculations to consider also execution trees starting with nodes containing several singletons. We will use similar notations. For example, $(BC)_i$ will denote the subtree of the full execution tree consisting of all nodes up to level i, starting with BC, and $E_{(BC)}(i)$ will denote the number of execution paths in $(BC)_i$.

Out of the four quantities above – $E_A(i), F_A(i), P_A(i), R_A(i)$ – only the last is really interesting for us. In fact, we are interested mostly in the asymptotics of $R_A(i)$ as $i \to \infty$. The average of these asymptotic values over all singletons A is the baseline (offline bound) we use to calculate the competitive ratio of online algorithms. To calculate $R_A(i)$, it will be convenient to start with the other three quantities. Moreover, as hinted above, it will be convenient to calculate these quantities not only for singletons, but for sets of singletons as well.

We start with the number of execution paths $E_{A_1A_2...A_k}(i)$, where $A_1A_2...A_k$ is any possible cache configuration and $i \geq 0$. We construct a system of recurrence equations for these quantities as follows. Suppose we want to express $E_{A_1A_2...A_k}(i+1)$ in terms of similar quantities, related to depth i. To this end, we need to open the node $A_1A_2...A_k$ just once for each of the possible combinations of requests. Thus, the node has $\prod_{j=1}^{n} |R_j|$ children. Each child $B_1B_2...B_h$ of $A_1A_2...A_k$ contributes $E_{B_1B_2...B_h}(i)$ to $E_{A_1A_2...A_k}(i+1)$. We obtain a system

of homogeneous linear recurrence equations with constant coefficients for the sequences $E_{A_1A_2...A_k}(i)_{i=0}^{\infty}$, over all possible cache configurations $A_1A_2...A_k$. The initial values $E_{A_1A_2...A_k}(0)$ are all 1. Note that all coefficients in the resulting equations are non-negative, and their sum in each equation is the same, namely $\prod_{j=1}^{n}|R_j|$. Thus, the matrix of coefficients is a scalar multiple of a stochastic matrix, which may facilitate the calculation.

The calculation of the $F_{A_1A_2...A_k}(i)$'s is similar, but the equations are this time non-homogeneous. That is, $F_{A_1A_2...A_k}(i+1)$ may be expressed as a sum of $F_{B_1B_2...B_h}(i)$'s, but we need to add also the number of page faults that are due to a 1-step expansion of $A_1A_2...A_k$. If this number is f, then the expression for $F_{A_1A_2...A_k}(i+1)$ contains an additional $f \cdot (\prod_{j=1}^{n}|R_j|)^i$ addend. Notice that the initial conditions this time are $F_{A_1A_2...A_k}(0)$ for all configurations $A_1A_2...A_k$.

Once $E_A(i)$ and $F_A(i)$ have been calculated, we readily find $P_A(i)$ and $R_A(i)$:

$$P_A(i) = \frac{F_A(i)}{E_A(i)}, \qquad R_A(i) = \frac{P_A(i)}{i}.$$

Recall that, as the recurrence equations defining $E_{A_1A_2...A_k}(i)$ and $F_{A_1A_2...A_k}(i)$ are linear with constant coefficients, it is easy to provide explicit expressions for them (assuming that one can find the eigenvalues of the matrix of coefficients, or at least good approximations of these eigenvalues). These expressions are combinations of exponentials (or exponential-polynomials if there are multiple eigenvalues). Hence their asymptotics is trivial to understand, and so is that of $R_A(i)$.

4 Offine algorithms to estimate the expected #pf

4.1 An optimal offline algorithm: full execution tree

In this section, we present a baseline for comparing online algorithms for scheduling multi-user concurrent distributed cache. This baseline is the optimal offline algorithm. This algorithm manages the cache in such a way that the average number of faults is minimal. Our first algorithm calculates the average accurately, for an arbitrary given distribution on the space of all request sequences, as long as this distribution results in an execution tree with a finite number of nonequivalent nodes. Usually, however, we will reduce the time this calculation requires by resorting to faster and less accurate algorithms. Our second and third algorithms will provide such simplifications.

In principle, to calculate #pf for the optimal algorithm, we need to run the algorithm for all possible request sequences, find the minimal number of page faults for each sequence, and then calculate the average over all sequences (under the assumed distribution over the space). This approach has indeed been taken in [7]. Below, we reduce the time required for the calculation by assigning some (unknown) number of future page faults to each state. We construct a tree, its nodes correspond to the possible states of the cache, and its edges correspond to the various possible requests at each state. Whenever we return to a state visited

earlier, we do not need to continue from this state anymore. We simplify the calculation further by noticing that various pairs of nodes of the tree are equivalent.

Example 1: Suppose we have two servers S_1, S_2, with corresponding sets of possible requests $R_1 = \{r_1^1, r_2^1\}$, $R_2 = \{r_1^2, r_2^2\}$, assumed to be disjoint, a cache of size $K = 3$, and access latency time $t = 0$. The $\binom{4}{3} = 4$ singletons are

$$A = \{r_1^1, r_2^1, r_1^2\},\ B = \{r_1^1, r_2^1, r_2^2\},\ C = \{r_1^1, r_1^2, r_2^2\},\ D = \{r_2^1, r_1^2, r_2^2\}.$$

At each time unit, there are 4 possibilities for the servers' requests, which we denote as follows:

$$\text{1-1} = (r_1^1, r_1^2),\ \text{1-2} = (r_1^1, r_2^2),\ \text{2-1} = (r_2^1, r_1^2),\ \text{2-2} = (r_2^1, r_2^2).$$

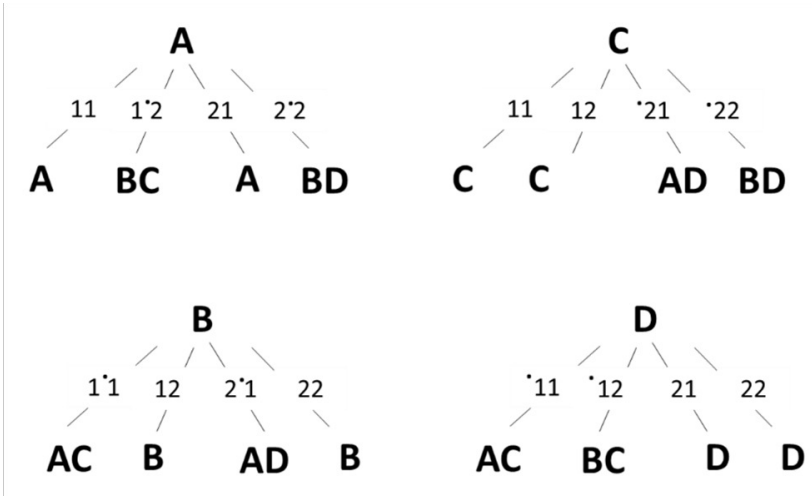

Fig. 1: A 1-step expansion of an execution tree for each singleton.

In Figure 1, we depict the 1-step expansions of all possible execution trees started by singletons. Here, a '.' label to the left of a request signifies that it yields a page fault.

Figure 2 provides 1-step expansions of the nodes comprising 2 singletons, obtained in Figure 1. An 'x' means that the corresponding branch is crossed out.

We have expanded all the possible cache configurations. Since all singletons are equivalent, a two-step execution tree started from any singleton, say from A, is as in Figure 3 .

It is easy to verify in this case that all singleton nodes are equivalent to each other, and so are all nodes consisting of a pair of singletons. We will use A as a

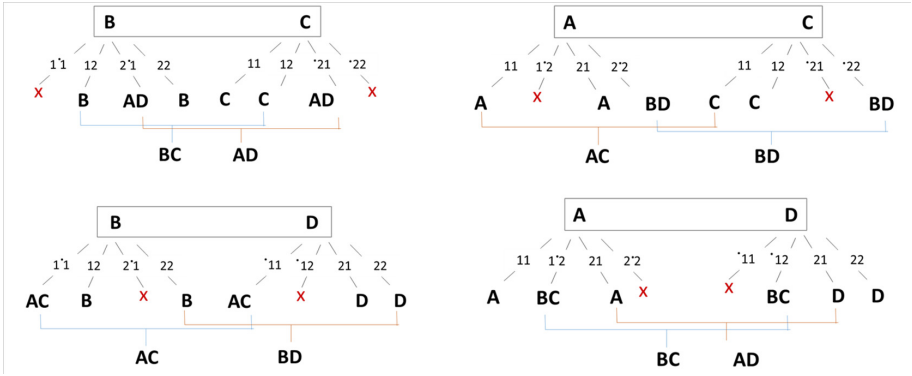

Fig. 2: A 1-step expansion of an execution tree for each generated pair of singletons.

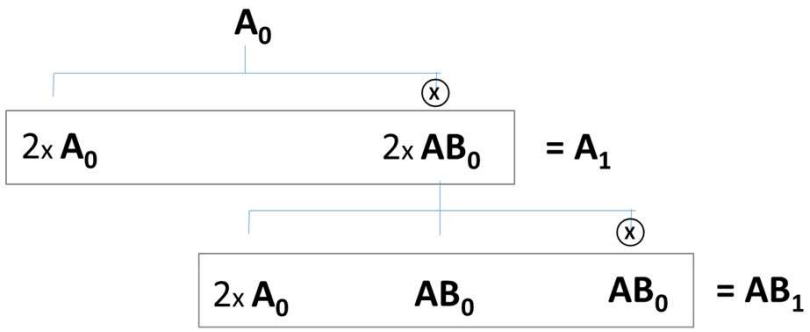

Fig. 3: A 2-steps expansion of an execution tree.

representative of the singletons, and AB as a representative pair. For the number of execution paths, we obtain the recursion:

$$\begin{aligned} E_A(i+1) &= 2E_A(i) + 2E_{AB}(i), \\ E_{AB}(i+1) &= 2E_A(i) + 2E_{AB}(i) \end{aligned}$$

Along with the initial conditions $E_A(0) = E_{AB}(0) = 1$, this yields:

$$E_A(i) = E_{AB}(i) = 4^i, \qquad i \geq 0.$$

For the number of page faults, the recurrence is:

$$\begin{aligned} F_A(i+1) &= 2F_A(i) + 2F_{AB}(i) + 2 \cdot 4^i, \\ F_{AB}(i+1) &= 2F_A(i) + 2F_{AB}(i) + 4^i \end{aligned}$$

The initial conditions are $F_A(0) = F_{AB}(0) = 0$. We obtain

$$F_A(i) = (\frac{3}{8}i + \frac{1}{4}) \cdot 4^i, \quad F_{AB}(i) = \frac{3}{8}i \cdot 4^i, \qquad i \geq 0.$$

It follows that

$$P_A(i) = \frac{(\frac{3}{8}i + \frac{1}{4}) \cdot 4^i}{4^i} = \frac{3}{8}i + \frac{1}{4}, \qquad i \geq 0,$$

and finally:

$$R_A(i) = \frac{3}{8} + \frac{1}{4i} \underset{i \to \infty}{\longrightarrow} \frac{3}{8}. \tag{1}$$

Thus, the expected #pf for the offline algorithm is 0.375, which is slightly more than one failure per 3 requests. Note that, by (1), the expected #pf per request reduces as we consider larger initial subtrees of the execution tree.

4.2 A simplified calculation of the asymptotics of R_A

In the preceding subsection, we explained how one can provide an exact formula for the average number of page faults per request in larger and larger initial subtrees of the full execution tree. In particular, we were able to calculate $\lim_{i \to \infty} R_A(i)$. In fact, this limit is usually the only measure of importance. In this subsection, we show how one can find the limit without calculating explicitly the whole sequence $R_A(i)_{i=0}^{\infty}$.

Note that one can view the process of moving from node to child in the tree as a Markov chain. (For a brief account of the basics of Markov chains, we refer the reader to [13] and [4].) The states of the chain consist in principle of all $2^{\binom{|R|}{K}}$ sets of cache configurations. We start from an arbitrary given singleton state. At each time unit, we get an n-tuple of requests from the n servers, and move to another state according to the mechanism explained in Section 3. We assume that each of the $\prod_{j=1}^{n} |R_j|$ possible vectors of requests occur with the same probability $1/\prod_{j=1}^{n} |R_j|$.

With this terminology, what we did in the preceding subsection was to find the probability of the chain being at any state at any given time, given the initial state. If indeed we only need the limiting probabilities, we only have to find the stationary probability vector. Note, that we do not need to deal with all the $2^{\binom{|R|}{K}}$ states. For example, we can never get to the empty set, and usually most states will be unreachable from any initial singleton state. Therefore, confine ourselves to the set of reachable states. Moreover, as in the preceding subsection, equivalent states may be merged.

The second option is to build an appropriate transition matrix which describes the given system, find its eigenvector corresponding to the eigenvalue 1, and then to calculate the expected #pf accordingly. We illustrate this technique in the following examples.

Example 2: We are given a system consisting of a single server, set $R = \{r_1, r_2, r_3\}$ of possible requests, cache of size $K = 2$, and access latency time $t = 0$. The possible configurations of the cache are $A = \{r_1, r_2\}$, $B = \{r_1, r_3\}$, and $C = \{r_2, r_3\}$. Without loss of generality, we start the execution from A. By symmetry all singletons are equivalent, as are all pairs of singletons. Hence, Figure 4 provides all the required information. A choice of a singleton from a pair is done using the FITF policy.

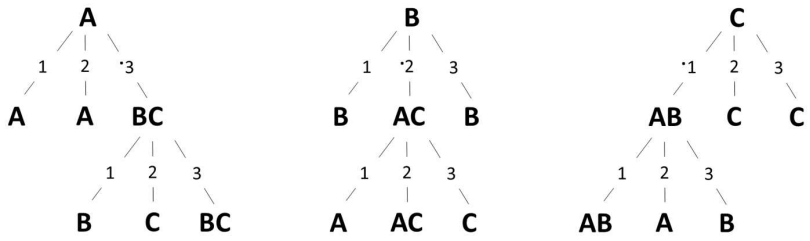

Fig. 4: The execution tree corresponding to Example 2.

From each singleton, there is a 2/3 probability of staying and a 1/3 probability of moving to a pair. The probabilities from a pair are the same. Hence, the matrix of probabilities is:

$$\begin{array}{c} \\ \text{singleton} \\ \text{pair} \end{array} \begin{array}{c} \begin{array}{cc} \text{singleton} & \text{pair} \end{array} \\ \begin{pmatrix} 2/3 & 1/3 \\ 2/3 & 1/3 \end{pmatrix} \end{array}$$

The rows of the matrix in this case are equal. Hence, the stationary probability is the same as these rows, namely $(2/3, 1/3)$. Thus, the expected #pf is $2/3 \cdot 1/3 + 1/3 \cdot 0 = 2/9$.

Example 3: We are given a system consisting of two servers S_1, S_2 with corresponding sets of possible requests $R_1 = \{r_1, r_2, r_3\}$, $R_2 = \{r_3, r_4\}$, a cache of size $K = 3$, and access latency time $t = 0$. The $\binom{4}{3} = 4$ singletons are:

$$A = \{r_1, r_2, r_3\},\ B = \{r_1, r_2, r_4\},\ C = \{r_2, r_3, r_4\},\ D = \{r_1, r_3, r_4\}.$$

At each time unit, there are 6 possibilities for the servers' requests, which we denote as follows:

$$\begin{array}{lll} \text{1-3} = (r_1, r_3), & \text{1-4} = (r_1, r_4), & \text{2-3} = (r_2, r_3), \\ \text{2-4} = (r_2, r_4), & \text{3-3} = (r_3, r_3), & \text{3-4} = (r_3, r_4). \end{array}$$

In Figure 5, we depict the 1-step expansions of all possible execution trees started by singletons.

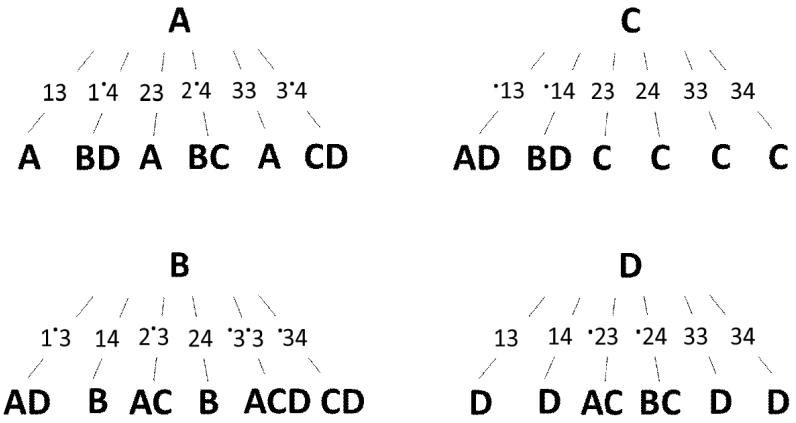

Fig. 5: A 1-step expansion of an execution tree for each singleton.

Figure 6 provides 1-step expansions of the nodes comprising 2 or 3 singletons, obtained in Figure 5.

The appropriate transition matrix is as follows:

$$\begin{array}{c} \\ A \\ B \\ C,D \\ BC,BD \\ CD \\ AC,AD \\ ACD \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C,D & BC,BD & CD & AC,AD & ACD \end{array} \\ \begin{pmatrix} 1/2 & 0 & 0 & 1/3 & 1/6 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 1/6 & 1/3 & 1/6 \\ 0 & 0 & 2/3 & 1/6 & 0 & 1/6 & 0 \\ 0 & 1/6 & 0 & 1/6 & 1/2 & 1/6 & 0 \\ 0 & 0 & 2/3 & 1/3 & 0 & 0 & 0 \\ 1/6 & 0 & 1/3 & 1/6 & 0 & 1/3 & 0 \\ 0 & 0 & 1/3 & 0 & 1/6 & 1/3 & 1/6 \end{pmatrix} \end{array}$$

The stationary probability vector is

$$p = \frac{1}{5595}(330, 260, 2296, 1040, 627, 990, 52) \approx$$
$$(0.058, 0.046, 0.41, 0.185, 0.112, 0.176, 0.009).$$

Therefore, the expected #pf is:

$$< p, (0.058, 0.046, 0.410, 0.185, 0.112, 0.176, 0.009) >= 0.266.$$

Note, that the eigenvalues of maximal modulus, below $\gamma_1 = 1$, are $\gamma_{2,3} = 0.369 \pm 0.185i$, with $|\gamma_{2,3}| = 0.413$. Thus, the error after i generations is $O(0.413^i)$.

4.3 Second offline algorithm: limited lookahead execution tree

When a full execution tree contains too many non-equivalent nodes, we may estimate #pf by expanding each singleton to some depth i. Then, we use one

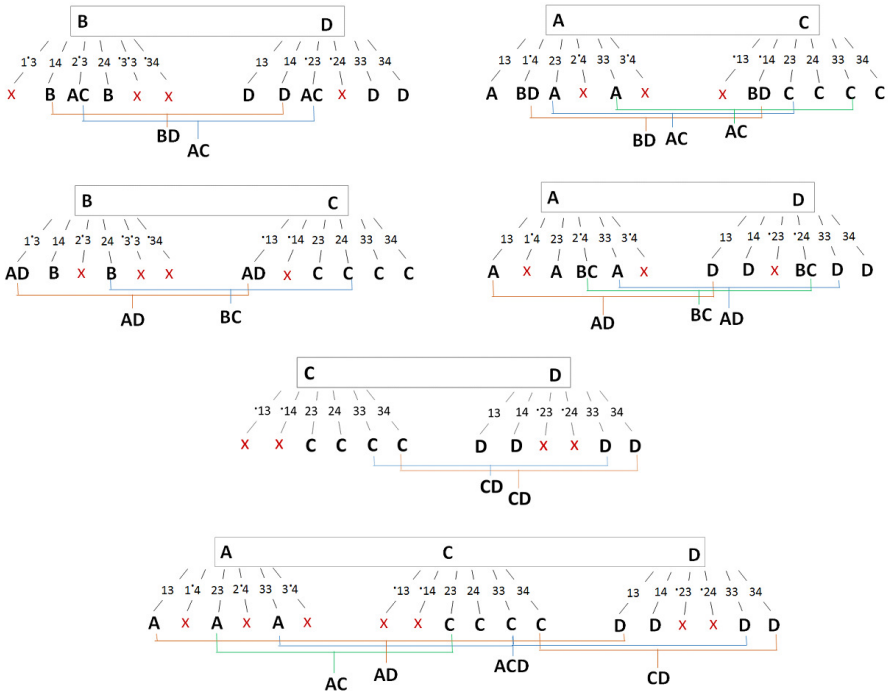

Fig. 6: A 1-step expansion of an execution tree for each generated pair and triple of singletons.

of the methods described above to calculate the expected #pf according to the obtained nodes. This will probably provide an upper bound on #pf, since the undiscovered nodes consist of several singletons, and thus tend to decrease the estimate on #pf.

Let us illustrate this approximate calculation on Example 3. Suppose we expanded all the singletons to depth $i = 1$. Thus, we have not checked the #pf for sets consisting of two or more singletons. The calculation of the expected #pf is as follows. There are 4 possible singletons to start an execution tree. Each of these leads to 6 children nodes. The total #pf from the initial nodes is $3+5+2+2 = 12$. Thus, the estimate for the expected #pf is $12/(4 \cdot 6) = 1/2$. We can see that this is actually an upper bound on the value 0.266 of the expected #pf we got for the same system in the previous section, using full expansion tree calculation. The size of i may vary according to available computing resources and the desirable execution speed. A larger value of i leads to a more precise estimate of #pf, but we would spend more time with calculating the expected #pf since each enlargement of i by 1 multiplies the number of leaves in the execution tree by $\prod_{j=1}^{n} |R_j|$.

5 Offline approximation algorithm to serve given request sequences: limited lookahead

In [7], an offline algorithm is presented which, given request sequences of the servers in the system, calculates the best way to serve them by trying all possibilities. The time complexity of this algorithm might be too high. We propose here a method for finding an approximation of the best way to serve the requests by using limited lookahead. The point is to limit the calculation of all possible ways of serving a given request sequence to some length i. We choose the path p minimizing the #pf for this length i. We repeat this process until all requests have been served.

There are two possibilities of progressing in such an execution tree. The first is to calculate a subtree of depth i, and execute all the steps of the seemingly optimal path found by this subtree. The second option is to find the same subtree, make only the first step of the obtained optimal path, and then recalculate the subtree of depth i once again. The second option is a compromise between finding deeper subtrees, which is much more expensive, and making some short-sighted and cheap steps. Recalculation of a subtree after each step increases the calculation time by i, while each enlargement of a subtree depth by 1 multiplies the calculation time by $\prod_{j=1}^{n} |R_j|$.

6 Heuristic online algorithm: dynamic machine learning

A simple-minded approach to the question of optimizing the cache is to put in it those requests that occur most often. Namely, suppose we did not have a cache. At each stage, we would have n page faults. (Recall that, if several servers ask simultaneously for the same r, we count each of them as causing an additional fault.) How much do we gain by placing some $r_j \in R$ in the cache? For each i such that $r_j \in R_i$, there is a probability of p_{ij} for S_i to ask for r_j (where $\sum_{j:r_j \in R_i} p_{ij} = 1$ for each i). Hence, by having r_j in the cache, we reduce #pf by $v(r_j) = \sum_{i:r_j R_i} p_{ij}$. To reduce #pf as much as possible, we place in the cache those requests r for which the quantity $v(r)$ is among the K maximal ones (ties broken arbitrarily). Note, that this tends to give more cache lines to servers with fewer possible requests.

Example 4: We are given a system with two servers S_1 and S_2, and their sets of possible requests $R_1 = \{r_1, r_2\}$, $R_2 = \{r_1, r_3, r_4\}$, respectively, where each server comes up with each of its possible requests with the same probability. Suppose the cache is of size $K = 2$. The average number of servers requesting each r_j at each time unit is:

$$
\begin{aligned}
v(r_1) &= 1/2 + 1/3 = 5/6, \\
v(r_2) &= 1/2, \\
v(r_3) &= 1/3, \\
v(r_4) &= 1/3,
\end{aligned}
$$

Thus, if we leave r_1 and r_2 in the cache permanently, we get the minimal #pf.

Lemma 1. *If the probabilities $p_{i,j}$ of each server S_i asking for each r_j do not change with time, and the servers are independent, then the algorithm proposed above is optimal.*

Proof. Denote the proposed algorithm by H, and let H' be some other algorithm. Denote by $C(H)$ the content of the cache when using H, and by $C(H')$ the analogous quantity for H' (which may change with time, depending on the way H' works). Then the expected number of page faults per time unit, when using H, is $\#\mathrm{pf}(H) = n - \sum_{r_j \in \, cache \ of \ H} v(r_j)$. The corresponding number for H' is $\#\mathrm{pf}(H') = n - \sum_{r_j \in \, cache \ of \ H'} v(r_j)$. Since the algorithm H chooses the j's with maximal $v(r_j)$, the number of page faults for H does not exceed the analogous number for H'. □

.

Given the above, we conclude that learning the recent request probabilities may lead to an efficient online heuristic cache management algorithm. We propose an algorithm which steps includes learning the optimal time window size ΔT and estimate the request probabilities. In the next time window, we use the algorithm described above. As long as the request probabilities stay put, we leave the cache unchanged.

The request probabilities typically change with time, both when certain processes terminate and as they proceed. To ensure that the algorithm is efficient, we need to be able to detect changes in the request probabilities, and change the cache contents accordingly. To this end, we need to learn the optimal size of time window we use. We may start with some default value of ΔT time units. At the end of the window we check whether a time window of $\Delta T/2, \Delta T/2^2, ..., \Delta T/2^k$ would yield smaller #pf. (Here, $\Delta T/2^k$ is some acceptable minimal window size.) If it turns out that one of these sizes outperforms ΔT, we replace ΔT by the seemingly best $\Delta T/2^j$. Of course, as time elapses, we may check whether windows of size $2\Delta T, 2^2\Delta T, ...$ may be better than the initial ΔT window.

Note, that our algorithm ensures that we do not fall in a "cold start" position, namely taking long-range decisions based on the behavior during a short time interval.

7 Future research directions

Recall that, in the case of a single server, LRU has an approximation ratio of $\Theta(K)$ in the worst case [9]. According to [7], in the case of concurrent usage, LRU may be arbitrarily worse than the offline algorithms. The most challenging question is whether one can design an online algorithm with a bounded approximation ratio relative to the optimal offline algorithm. If so, how can we reach the best ratio?

Acknowledgments

This research is partially supported by EMC; the Milken Families Foundation Chair in Mathematics; the Rita Altura Trust Chair in Computer Sciences; grant of the Ministry of Science, Technology and Space, Israel, and the National Science Council (NSC) of Taiwan; the Ministry of Foreign Affairs, Italy; the Ministry of Science, Technology and Space, Infrastructure Research in the Field of Advanced Computing and Cyber Security and the Israel National Cyber Bureau.

References

1. B. Awerbuch, Y. Bartal, and A. Fiat, "Distributed paging for general networks", *In Proc. of the 7th Ann. ACMSIAM Symp. on Discrete Algorithms, 574–583,* 1996.
2. A. Borodin and R. El-Yaniv, "Online Computation and Competitive Analysis", *vol. 53. Cambridge University Press, New York,* 1998.
3. J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors" *In ICS '07: Proceedings of the 21st annual international conference on Supercomputing, 242–252, New York, NY, USA, ACM* 2007.
4. E. B. Dynkin, "Theory of Markov Processes", *Pergamon,* 1960.
5. A. Hassidim, "Cache replacement policies for multicore processors", *In Proceedings of 1st Symposium on Innovations in Computer Science (ICS), 501–509,* 2010.
6. E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches" *In Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures, 11–22,* 2008.
7. A. Lopez-Ortiz and A. Salinger, "Paging for multi-core shared caches", *In Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, 113–127,* 2012.
8. A. M. Molnos, S. D. Cotofana, M. J. M. Heijligers, and J. T. J. van Eijndhoven, "Throughput optimization via cache partitioning for embedded multiprocessors" *In G. Gaydadjiev, C. J. Glossner, J. Takala, and S. Vassiliadis, editors, ICSAMOS, 185–192. IEEE,* 2006.
9. D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules", *Communications of the ACM, 28(2):202–208,* 1985.
10. S. Srikantaiah, M. Kandemir, and Q. Wang, "Sharp control: controlled shared cache management in chip multiprocessors" *In MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 517–528, New York, NY, USA, ACM.* 2009.
11. H. Stone, J. Turek, and J. Wolf, "Optimal partitioning of cache memory" *IEEE Transactions on Computers, 41:1054–1068,* 1992.
12. G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic cache partitioning for simultaneous multithreading systems" *In Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, 116–127,* 2001.
13. H. M. Taylor and S. Karlin, "An Introduction to Stochastic Modeling", *San Diego: Academic Press, 3rd ed.,* 1998.