Brief Announcement: Multicore Paging Algorithms Cannot Be Competitive

Shahin Kamali
shahin.kamali@umanitoba.ca
Department of Computer Science
University of Manitoba
Winnipeg, Mb, Canada

Helen Xu
hjxu@mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA

ABSTRACT

Every processor with multiple cores sharing a cache needs to implement a page-replacement algorithm. López-Ortiz and Salinger [5] demonstrated that competitive ratio of canonical paging algorithms such as Least-Recently-Used (LRU) and Furthest-In-Future (FIF) grows with the length of the input. In this paper, we answer an open question about the existence of competitive multicore paging algorithms in the negative. Specifically, we show that all lazy algorithms, which include all practical algorithms, cannot be competitive against the optimal offline algorithm.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**.

KEYWORDS

Multicore paging, caching, parallel architectures, online algorithms

ACM Reference Format:
Shahin Kamali and Helen Xu. 2020. Brief Announcement: Multicore Paging Algorithms Cannot Be Competitive. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400270

1 INTRODUCTION

Despite the widespread use of multiple cores in a single machine, the theoretical performance of even the most common cache eviction algorithms is not yet fully understood when multiple cores simultaneously share a cache. Paging algorithms for multicore architectures have been well-studied in practice, including dynamic cache-partitioning heuristics [8, 10, 11] and operating system cache management [2, 7, 12]. There are very few theoretical guarantees, however, for performance of these algorithms.

In this paper, we explore the ***multicore paging problem*** in which multiple cores share a cache and request pages in an online manner. Upon serving a request, the requested page should become available in the shared cache. If the page is already in the cache, a hit takes place; otherwise, when the page is not in the cache, the core that issues the request incurs a fault. In case of a fault, the requested page should be fetched to the cache from a slow memory. Fetching a page causes a ***fetch delay*** in serving the subsequent requests made by the core that incurs the fault. Such delay is captured by the free-interleaving model of multicore paging [4, 5]. Under this model, when a core incurs a fault, it spends multiple cycles fetching the page from the slow memory while other cores may continue serving their requests in the meantime. Therefore, an algorithm's eviction strategy not only defines the state of the cache and the number of faults, but also the order in which requests are served. That is, a paging algorithm implicitly defines a "schedule" of requests served at each timestep through its previous eviction decisions.

Divergence between multicore and single-core paging. López-Ortiz and Salinger [5] leveraged the scheduling aspect of multicore paging to demonstrate that guarantees on competitive ratio[1] of algorithms in the single-core setting do not extend to multicore paging. In particular, they focused on two classical single-core paging algorithms, Least-Recently-Used (LRU) [9] and Furthest-In-Future (FIF) [1], and showed these algorithms are unboundedly worse than the optimal algorithm OPT in the multicore setting. LRU is an online paging algorithm that evicts the least-recently-requested page. FIF is an offline paging algorithm that evicts the page that will be requested furthest in the future. In the single-core setting, LRU is k-competitive (where k is the size of the cache) [9], and FIF is the optimal algorithm [1].

Non-competitiveness of lazy algorithms. We confirm the intuition that multicore paging is much harder than single-core paging and show that all practical ***lazy*** algorithms are equivalently non-competitive [2] against OPT (Corollary 3.2). More precisely, we provide adversarial inputs formed by a total of n requests that show the competitive ratio of any lazy algorithm is $\Omega(n^{1/2}/k)$.

An online algorithm is lazy[3] if it 1) evicts a page only if there is a fault 2) evicts at most one page in case of a fault, 3) for all timesteps, does not evict a page that incurred a hit in that timestep, and 4) evicts a page only if there is no space left in the cache. Algorithms with properties 1-3 (but not necessarily 4) are called "honest" algorithms [5]. Lazy algorithms capture natural properties of online algorithms. For example, if there was a hit on a page σ at some timestep, a lazy algorithm does not evict σ in that same timestep. Additionally, once the cache is full, a lazy algorithm keeps it full. Common paging strategies such as LRU and First-In-First-Out (FIFO) are clearly lazy.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400270

[1] For a cost-minimization problem, an online algorithm has a competitive ratio of at most c if its cost on any input never exceeds c times the cost of an optimal offline algorithm (up to an additive constant).
[2] A multicore paging algorithm is not competitive if its competitive ratio depends on n, the length of the input.
[3] We adopt the k-server definition of lazy algorithms [6].

2 PROBLEM DEFINITION

This section reviews the free-interleaving model [4, 5] of multicore paging.

Assume we are given a multicore processor with p cores labeled $P_1, P_2, \ldots, P_p$ and a shared cache with k pages ($k \gg p$).

Input description. An input to the ***multicore paging problem*** is formed by p request sequences $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_p)$. Each core P_i must serve its corresponding ***request sequence*** $\mathcal{R}_i = \langle \sigma_{i,1}, \ldots, \sigma_{i,n_i} \rangle$ made up of n_i page requests. For all i, we assume $n_i \gg k, \tau$. The total number of page requests is therefore $n = \sum_{1 \le i \le p} n_i$.

Free-interleaving model. Page requests arrive at discrete timesteps. The requests issued by each core should be served in the same order that they appear and in an online manner. More precisely, for all $i, j \ge 1$, core P_i must serve request $\sigma_{i,j}$ before $\sigma_{i,j+1}$, and $\sigma_{i,j+1}$ is not revealed before $\sigma_{i,j}$ is served. The multicore processor may serve at most p page requests in parallel (up to one request per core[4]). Each page request must be served as soon as it arrives. To serve a request to some page $\sigma_{i,j}$ in sequence $\mathcal{R}_i$, core P_i either has a ***hit***, when $\sigma_{i,j}$ is already in the cache, or incurs a ***fault*** when $\sigma_{i,j}$ is not present in the cache. In case of a fault, the requested page should be fetched into the cache. It takes τ timesteps to fetch a page into the cached, where τ is an integer parameter of the problem. During these timesteps, P_i cannot see any of its forthcoming requests, that is, $\sigma_{i,j+1}$ is not revealed to P_i before $\sigma_{i,j}$ is ***fully fetched***. In case some other core $P^* \neq P_i$ is already fetching the page when the fault occurs, P_i waits for less than τ timesteps until the page is fully fetched to the cache. We measure the cost of algorithms in terms of the *number of faults* they incur while serving an input.

A multicore paging algorithm $\mathcal{A}$ reads requests from request sequences in parallel and is defined by its eviction decisions at each timestep. If a core faults while the cache is full, $\mathcal{A}$ must evict a page to make space for the requested page before fetching it. We continue the convention [3, 5] that when a page is evicted, the cache cell that previously held the evicted page is unused until the replacement page is fetched. Finally, the processor serves requests from different request sequences in the same timestep in some fixed order (e.g., by core index).

3 NON-COMPETITIVENESS OF LAZY ALGORITHMS

We show that no lazy algorithm for multicore paging is competitive. We first show that no algorithm is competitive for the case of two cores in Theorem 3.1, then extend our argument to an arbitrary number of cores in Corollary 3.2.

Theorem 3.1. *When there are $p = 2$ cores, the competitive ratio of any lazy algorithm $\mathcal{A}$ is $\Omega(n^{1/2}/k)$ in terms of the number of faults.*

Proof Sketch. The adversary constructs an input formed by requesting pages from two disjoint sets of k "red" (r) and k "blue" (b) pages over ϕ rounds ($\phi \approx \sqrt{n}/2$). The requests made by each core in each round forms an "easy phase" followed by a "hard phase". Each phase is formed by exactly ℓ requests ($\ell \approx \sqrt{n}$).

Each phase has a color that is associated with the color of most requested pages in that phase. Easy and hard phases of P_1 are all respectively blue and red, while easy and hard phases of P_2 are all respectively red and blue. With the exception of the first phase of P_1 and the last phase of P_2, any phase in each core gets a "partner" phase of the same color in the other core. Specifically, the ith easy phase of P_1 is partnered with the $(i-1)$-th hard phase of P_2, and the ith easy phase of P_2 is partnered with the ith hard phase of P_1.

The adversary defines an input such that there exists an (honest but not lazy) offline algorithm OFF that serves the requests in partner phases at the same time. Figure 1 illustrates the alignment of the two cores in $\mathcal{A}$ and OFF as well as the cache state in selected timesteps.

Next, we formalize how the adversary makes requests in each phase. Every phase has exactly ℓ page requests. Easy phases are formed by requests to only two pages. The first phase of P_1 differs from the rest of the easy phases because it does not have a partner and requests two red pages in a loop. All later easy phases are defined based on the decisions of $\mathcal{A}$ during their partner hard phases. Let $Q_{e,i}$ be the ith easy phase of color $c \in \{r, b\}$ and let $q_i^c \le k$ denote the number of requests to unique pages made in $Q_{e,i}$'s corresponding hard phase $Q_{h,i}$. For all i, the adversary generates $Q_{e,i}$ by repeating requests to two pages of different colors followed by requests to one page of the same color of the phrase. The initial two pages are arbitrarily selected from the set of red/blue pages in the $\mathcal{A}$'s cache just before the start of $Q_{e,i}$. At the beginning of $Q_{e,i}$, the two red/blue pages are requested one after the other for the first q_i^c requests, while the remaining $\ell - q_i^c$ requests are to the single page that has color c. The two pages that form the requests of an easy phase are always in the cache of $\mathcal{A}$ at the beginning of the phase. Moreover, $\mathcal{A}$ serves the easy phases of P_1 at the same time as easy phases of P_2 and incurs all hits for both cores (with the exception of the cold misses in the first easy phase).

Hard phases are made by requesting the pages that are absent from $\mathcal{A}$'s cache. At the beginning of any hard phase, at least one page of each color is present in the cache of $\mathcal{A}$. Because there are k pages of each color, the adversary generates the hard phase of a given color by always asking for some absent page of that color and $\mathcal{A}$ incurs a fault on every request of a hard phase. The inductive construction ensures the ith request of the two cores are served at the same time. In particular, the hard phases start at the same time and hence there will be congestion in the cache during the hard phases. The number of faults by $\mathcal{A}$ is at least equal to the total length of hard phases. There are $\Theta(\sqrt{n})$ hard phases, each of length $\Theta(\sqrt{n})$, for a total cost of $\mathcal{A}(\mathcal{R}) = \Theta(n)$.

An offline algorithm OFF can schedule the input such that partner easy and hard phases are served at the same time. This is done by "postponing" the first easy red phase (the first phase of P_2) by always evicting the page of the phase that is present in the cache before fetching the other page of the phase. Meanwhile, OFF hits on all requests in the first easy blue phase (the first phase of P_1) except the first two cold misses (it simply brings the two pages into the cache). After the first easy blue phase ends, OFF serves the remainder of the first easy red phase together with its partner, the first red hard phase, in a way that they end at the same time. This

[4] In practice, a single instruction of a core may involve more than one page, but we assume that each request is to one page in order to model RISC architectures with separate data and instruction caches [5].

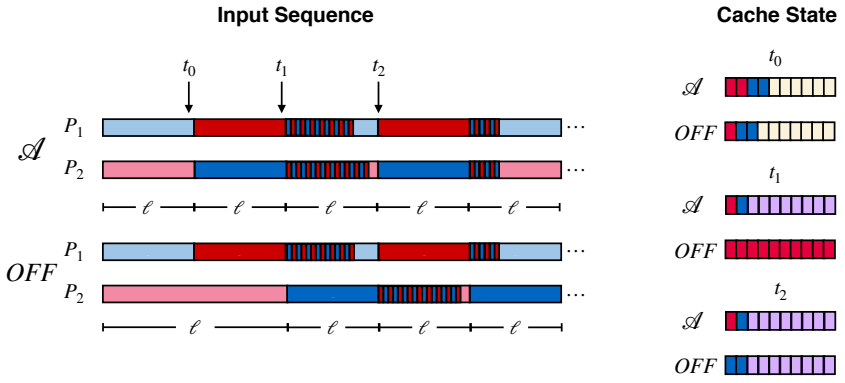

Figure 1: An example of the alignment of $\mathcal{A}$ and OFF described in Theorem 3.1 of two cores. For each sequence, easy phases consist of cycles of requests to two distinct pages (stripes of red and blue) followed by requests to a single page (light red and light blue), while hard (dark red and dark blue) phases are adversarial and designed so $\mathcal{A}$ faults on every request.
Left: $\mathcal{A}$ serving all cores in their easy and hard phases at the same time while OFF delays P_2.
Right: Examples of the cache state with $k = 10$ at each of the timesteps marked t_0, t_1, t_2. At t_0, the yellow cells represent empty cells. At t_1 after the first red hard phase, $\mathcal{A}$ also finishes its first blue hard phase while OFF finishes the first blue easy phase. The light purple cells are those that could either be red or blue. At timestep t_2, $\mathcal{A}$ has at least one of each color page in its cache, whereas OFF might only have blue pages.

scheme also applies for other phases, i.e., any phase starts and ends with its partner. To maintain this alignment, OFF ensures there is a fault in the easy phase for each fault in the hard phase. When a page is requested in the hard phase for the first time, OFF keeps it in the cache until the end of the phase. Therefore, there is a fault in a hard phase only when a page is requested for the first time during the phase. The first q_i^c requests of the partner of the phase (an easy phase) are to two different pages of different colors. Upon a fault in the hard phase, OFF evicts one of these pages. This ensures the next request is a fault in the easy phase. In particular, in the very last fault of the hard phase (after q_i^c requests), OFF evicts the page that has color other than c. Consequently, in the remainder of these two partnered phases, all requested pages are in the cache and OFF hits on all of the remaining requests.

Analysis. The total number of faults by OFF in the two partner phases will be no more than $2k$ (up to k for each). There are $\phi - 1 = \Theta(\sqrt{n})$ pairs of partner phases for a total of $\Theta(k\sqrt{n})$ faults. The first phase of P_1 and the last phase of P_2 have length ℓ and OFF incurs no more than $\ell = \Theta(\sqrt{n})$ faults in them. In conclusion, the total number of faults in $\mathcal{A}$ and OFF are respectively $\Theta(n)$ and $\Theta(k\sqrt{n})$, which gives a competitive ratio of $\Omega(n^{1/2}/k)$ for $\mathcal{A}$. □

For the case of arbitrary $p > 2$, we can partition the cores into two disjoint groups, assign to the two colors, and repeat the input from Theorem 3.1 across cores to get the following corollary.

Corollary 3.2. *For any number of cores, the competitive ratio of any lazy algorithm $\mathcal{A}$ is $\Omega(n^{1/2}/k)$.*

4 CONCLUSIONS

We showed that no lazy algorithm is competitive because the adversary has the power to artificially delay sequences. The scheduling power of OPT in multicore paging motivates the need for alternative measures of online algorithms.

ACKNOWLEDGEMENTS

Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.
[2] Alexandra Fedorova, Margo I. Seltzer, and Michael D. Smith. 2006. *Cache-Fair Thread Scheduling for Multicore Processors.* Technical Report. Harvard University.
[3] Avinatan Hassidim. 2010. Cache Replacement Policies for Multicore Processors.. In *ICS*. 501–509.
[4] Anil Kumar Katti and Vijaya Ramachandran. 2012. Competitive Cache Replacement Strategies for Shared Cache Environments. In *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. 215–226.
[5] Alejandro López-Ortiz and Alejandro Salinger. 2012. Paging for Multi-Core Shared Caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 113–127.
[6] Mark S Manasse, Lyle A McGeoch, and Daniel D Sleator. 1990. Competitive algorithms for server problems. *Journal of Algorithms* 11, 2 (1990), 208–230.
[7] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *ACM SIGARCH Computer Architecture News*, Vol. 35. 381–391.
[8] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Microarchitecture,MICRO. ACM International Symposium On*. 423–432.
[9] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208.
[10] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal Partitioning of Cache Memory. *IEEE Transactions on computers* 41, 9 (1992), 1054–1068.
[11] G Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28, 1 (2004), 7–26.
[12] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 174–183.