Brief Announcement: Green Paging and Parallel Paging

Kunal Agrawal
Washington University in St. Louis
St. Louis, MO, USA
kunal@wustl.edu

Michael A. Bender
Stony Brook University
Stony Brook, NY, USA
bender@cs.stonybrook.edu

Rathish Das
Stony Brook University
Stony Brook, NY, USA
radas@cs.stonybrook.edu

William Kuszmaul
MIT
Cambridge, MA, USA
kuszmaul@mit.edu

Enoch Peserico
Univ. Padova
Padova, Italy
enoch@dei.unipd.it

Michele Scquizzato
University of Padova
Padova, Italy
scquizza@math.unipd.it

ABSTRACT

We study two fundamental variants of the classic paging problem: *green paging* and *parallel paging*. In green paging one can choose the exact memory capacity in use at any given instant, between a maximum of k and a minimum of k/p pages; the goal is to minimize the integral of this number over the time required to complete a computation (note that running at lower capacity is not necessarily better, since might disproportionately increase the total completion time). In parallel paging, a memory of k pages is shared between p processors, each carrying out a separate computation; the goal is to minimize the respective completion times.

We show how these two different problems are strictly related: any efficient solution to green paging can be converted into an efficient solution to parallel paging, and any lower bound for green paging can be converted into a lower bound for parallel paging—in both cases in a black-box fashion. Exploiting this relation, we provide tight upper and lower bounds of $\Theta(\log p)$ on the competitive ratio with $O(1)$ resource augmentation for both problems.

CCS CONCEPTS

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

KEYWORDS

Paging; online algorithms; green computing; shared cache

ACM Reference Format:
Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2020. Brief Announcement: Green Paging and Parallel Paging. 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/10.1145/3350755.3400231

1 INTRODUCTION

The memory system of computing devices is typically organized as multiple layers of progressively larger capacity but also higher access cost (in terms of both time and energy); efficiently orchestrating the flow of information across memory layers is crucial for performance. The most widely used model for studying this classic *paging problem* is that of a two-layer system: a smaller *memory* layer with a capacity of k *pages*, and a larger layer of infinite capacity whose pages can only be accessed by first copying them into memory—an operation called (servicing a page) *fault*.[1] Given any sequence of pages that must be accessed in order, a paging algorithm chooses which page(s) to evict from memory, whenever a new page must be copied into it, so as to minimize the total number of faults.

The simple algorithm LFD (Longest Forward Distance) that evicts the page accessed furthest in the future has long been known to be optimal [2, 16]. However, paging is often studied as an *online* problem, i.e., an algorithm can decide evictions only on the basis of past requests. The typical framework for evaluating the performance of online algorithms is that of *competitive analysis* [21]. A paging algorithm is said to have a *competitive ratio* of (no more than) ρ if, for every request sequence, it incurs at most ρ times as many faults as an optimal offline algorithm incurs with a memory of capacity $h \leq k$ (plus an additive constant independent of the sequence length). The ratio k/h, called *resource augmentation*, and competitive ratio can then be seen as the space and time overheads of servicing a sequence of page requests online, i.e. without knowing the future.

In the competitive analysis framework, the classic paging problem is very well-understood: many simple algorithms including LRU, FIFO, FWF, and CLOCK have a competitive ratio of $\frac{k}{k-h+1}$ [4, 21], optimal for deterministic algorithms. In the rest of this section we present and motivate two extensions of classic paging, *green paging* and *parallel paging*; we then follow with a brief overview of our results and their implications.

1.1 Green Paging

In *green paging* we allow memory capacity to vary over time under the control of the paging algorithm, between a maximum of k and a minimum of k/p pages. Accessing a page in memory takes 1 unit of time, while a page fault takes $s \gg 1$ units (the *full access cost* model of [25]). The goal is to minimize, rather than the total time/faults

[1] We follow the historical convention of using the terms "memory", "pages", and "fault" to refer to a generic memory layer, its blocks, and an access whereby a block is fetched from the next larger layer—e.g. the processor cache, its lines, and a cache miss. Sometimes these two layers are referred to as *fast memory* and *slow memory*, respectively.

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

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400231

taken to service a sequence of page requests, the integral of memory capacity over that time.

The last decade has seen a surge in interest for paging models where memory capacity can change over time. One basis for such models lies in the popularity of virtualization/cloud computing services, that allow one to rent computational resources on demand; minimizing the integral of memory capacity "rented" for a computation minimizes monetary cost. Another basis for such models lies in the increasing importance of energy consumption for both mobile and supercomputing platforms: modern hardware can dynamically turn off portions of the memory, both at the main memory and processor cache layers, so that instantaneous power consumption is proportional to the amount of active memory—and total energy consumption is proportional to its integral over time. It is crucial to observe that minimizing power by minimizing active memory does not necessarily minimize *energy*, i.e. the integral of power over time, since less memory may yield disproportionately longer executions. Also, note that below a certain capacity, other costs may become dominant; hence our choice of a minimum capacity below which no substantial savings can be realized.

The first to address a similar problem was Chrobak [6], allowing the paging algorithm to determine both the capacity and the contents of the memory on any given request, with the goal of minimizing a linear combination of the total number of faults and the average capacity over all requests. This problem has been investigated by López-Ortiz and Salinger [14] and later, in the more general version where pages have sizes and weights, by Gupta et al. [10]. It turns out [3, 18, 19] that one can effectively decouple page replacement from memory allocation: even if the latter is chosen adversarially, LFD is still optimal, and a number of well-known paging algorithms like LRU or FIFO sport $O(1)$ competitive ratios with $O(1)$ resource augmentation (as in classic paging). Thus, green paging is essentially a problem of memory allocation: once memory is allocated, one can simply use LRU for page replacement, as it will incur a cost within a factor of $O(1)$ of what is achievable with (half) that memory capacity.

1.2 Parallel Paging

Another crucial extension of classic paging involves p processors sharing the same k-page memory. Each processor runs its own application, and the set of pages accessed by different applications are disjoint. Again, accessing a page in memory takes 1 unit of time, while a page fault takes $s \gg 1$ units. The goal is to service the request sequences minimizing the average completion time, or the completion time of all sequences but a fraction ϵ (with $\epsilon = 1/2$ and $\epsilon < 1/p$ yielding, respectively, median and maximum completion time).

Parallel paging has been extensively studied within the systems community, particularly after multicore processors became mainstream—starting from some pioneering work on heuristics that dynamically adjust the sizes of the cache partitions dedicated to each processor core (see, e.g., [22–24]). Parallel paging introduces several challenges compared to the classic problem. Multiple processors compete for the same resource, and the paging algorithm must decide, for each processor and each time, how many and which of its pages to keep in cache [7, 11, 15]. The marginal benefit of more memory may vary across processors and the resulting optimization problem is in general non-convex; and since this benefit can vary over time a paging algorithm should change the number of memory pages allocated to the processors accordingly. Furthermore, providing more space to some processors might accelerate the respective computations, and thus change how all computations are synchronized and the respective accesses interleaved. This can have a significant albeit often overlooked effect on performance: intuitively, we would like to synchronize computations so that portions with the most "compatible" memory usage run at the same time.

Almost all previous theoretical work models the problem as a straightforward variant of classic paging: given p disjoint request sequences (one per processor) interleaved a priori into a single sequence, one must service the latter choosing which pages to keep in memory so as to minimize the total number of faults [1, 5, 9, 12, 13] or other metrics [17]. The only complication compared to classic paging may be the choice of the interleave pattern, as in [8]. Crucially, this "fixed interleave" model effectively assumes that, when a processor incurs a fault, all others remain idle until the fault is resolved. But this negates the very premise on which parallel processing is based: when a processor encounters a fault, the other processors should continue working. The recent [11, 15] are the only two works assuming that, while a processor is blocked on a fault, others can advance. They investigate the complexity of the offline problem and show lower bounds for traditional paging algorithms such as LRU. However, no competitive online algorithms are given, leaving our understanding of parallel paging largely incomplete.

2 OUR RESULTS

This work (based partly on some preliminary results in [20]) analyzes both green paging and parallel paging: a surprisingly strict relationship between the two allows us to translate results for one problem into results for the other. In this sense we stress that the use of the same variable p for apparently unrelated quantities in the two problems is intentional, as it plays exactly the same role. This tight relationship has a counter-intuitive implication. For decades, little progress was made on parallel paging partly because it was unclear how to handle the interleaving and interference between different processors (see, e.g., [11]). Our equivalence results show that each processor can in some sense be handled in isolation!

Two fundamental ingredients in our construction are worth mentioning. The first is a simple but powerful technique to cut up memory allocation into a sequence of boxes chosen from a small set of "standard" sizes. While a similar technique was used e.g. in [3] as a tool to simplify analysis, we actively use it to allocate memory – our performance hinges crucially on the set of standard boxes being small. The second ingredient, used in the translation from green to parallel paging, is a way to pack these boxes so as to use available space and time efficiently; note that standard packing techniques cannot in general be used, in that memory boxes belonging to the same processor must be allocated to disjoint time intervals.

2.1 From Green to Parallel, and Back

Good green paging yields good parallel paging. We show how to translate *any* green paging algorithm into a parallel paging

algorithm in a black box fashion. If the former is online, so is the latter. If the former is optimal within a factor α when servicing request sequences from an arbitrary family, the latter sports average completion time that is optimal within a factor $O(\alpha)$ when servicing sequences from the same family, and it completes all sequences save at most a fraction ϵ taking at most $O(\alpha \log(\epsilon^{-1}))$ more time than what is strictly required to service all but a fraction $\epsilon/2$.

We would stress two things. First, the *same* translation yields simultaneously good average, median ($\epsilon = 1/2$), and maximum ($\epsilon < 1/p$) times, i.e. there is no need to sacrifice good performance under one metric for good performance under another. Second, note that if a green paging sports particularly good performance on a restricted family of request sequences of interest, perhaps bypassing the lower bounds below, we automatically obtain good performance for parallel paging on those same sequences.

The reverse is also true. We show how to translate *any* parallel paging algorithm into a green paging algorithm in a black box fashion. If the former is online, so is the latter. If the former has average or median completion time optimal within a factor $O(\alpha)$, so has the latter, provided the former is either a) offline or b) "fair", in the sense that if servicing identical request sequences it allots the same amount of memory to each at the same point in the execution.

The fairness requirement makes efficient translation not as universal as in the previous case. In particular, it does not allow us to translate lower bounds on green paging to lower bounds for parallel paging (in principle they may not apply to "unfair" online algorithms). However, we can show how to translate such lower bounds directly: if any online algorithm for green paging has a competitive ratio at least α even when restricted to servicing sequences from a particular family, then any online parallel paging algorithm has competitive ratio at least $\Theta(\alpha)$ even when restricted to sequences from the same family.

2.2 Tight Bounds for Green and Parallel Paging

We derive tight lower and upper bounds for the competitive ratio of green paging. We show that with $O(1)$ resource augmentation, no green paging can be better than $O(\log p)$-competitive (in contrast with classic paging, at least if $p = \omega(1)$). And this lower bound can be matched by a simple memory allocation algorithm that is not only online, but also memoryless, i.e., it does not depend in any way on the request sequence.

Note that the bounds above depend solely on the ratio p between the maximum and minimum memory, and not on those two values k and k/p. Thus, if $p = O(1)$, we automatically have $O(1)$-competitive green paging (with $O(1)$ resource augmentation). Also, note that the lower bound is existential; for some restricted family of sequences of practical interest it may well not apply.

The results from the previous subsection automatically allow us to translate these bounds and algorithms to parallel paging. No parallel paging algorithm can be better than $O(\log p)$-competitive. At the same time, we exhibit a simple strategy for dividing memory between processors *that is completely independent of the request sequences to be serviced* and that is nonetheless (optimally) $O(\log p)$-competitive in terms of average and median completion time, and $O(\log^2 p)$-competitive in terms of maximum completion time.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, CCF-1439084, CCF-1733873, and CCF-1527692; by the US Air Force Research Laboratory under cooperative agreement number FA8750-19-2-1000; and by Univ. Padova under grant BIRD197859/19 and project "Internet of Things" (MIUR grant L.232 "Dipartimenti di Eccellenza").

REFERENCES

[1] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. 2000. Application-controlled paging for a shared cache. *SIAM J. Comput.* 29, 4 (2000), 1290–1303.

[2] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[3] Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-Adaptive Algorithms. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 958–971.

[4] Allan Borodin, Nathan Linial, and Michael E. Saks. 1992. An Optimal On-Line Algorithm for Metrical Task System. *J. ACM* 39, 4 (1992), 745–763.

[5] Pei Cao, Edward W. Felten, and Kai Li. 1994. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)* (Boston, Massachusetts). 171–182.

[6] Marek Chrobak. 2010. SIGACT news online algorithms column 17. *SIGACT News* 41, 4 (2010), 114–121.

[7] Rathish Das, Kunal Agrawal, Michael Bender, Jonathan Berry, Benjamin Moseley, and Cynthia Phillips. 2020. How to Manage High-Bandwidth Memory Automatically. In *Proceedings of the 32st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[8] Esteban Feuerstein and Alejandro Strejilevich de Loma. 2002. On-Line Multi-Threaded Paging. *Algorithmica* 32, 1 (2002), 36–60.

[9] Amos Fiat and Anna R. Karlin. 1995. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th annual ACM Symposium on Theory of Computing (STOC)*. 626–634.

[10] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. 2019. Elastic Caching. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 143–156.

[11] Avinatan Hassidim. 2010. Cache Replacement Policies for Multicore Processors. In *Proceedings of 1st Symposium on Innovations in Computer Science (ICS)*. 501–509.

[12] Anil Kumar Katti and Vijaya Ramachandran. 2012. Competitive Cache Replacement Strategies for Shared Cache Environments. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 215–226.

[13] Ravi Kumar, Manish Purohit, Zoya Svitkina, and Erik Vee. 2020. Interleaved Caching with Access Graphs. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1846–1858.

[14] Alejandro López-Ortiz and Alejandro Salinger. 2012. Minimizing Cache Usage in Paging. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*. 145–158.

[15] Alejandro López-Ortiz and Alejandro Salinger. 2012. Paging for Multi-Core Shared Caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science conference (ITCS)*. 113–127.

[16] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117.

[17] Ishai Menache and Mohit Singh. 2015. Online Caching with Convex Costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 46–54.

[18] Enoch Peserico. 2013. Elastic paging. In *Proceedings of the ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 349–350.

[19] Enoch Peserico. 2019. Paging with dynamic memory capacity. In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science (STACS)*. 56:1–56:18.

[20] Michele Scquizzato. 2013. *Paging on Complex Architectures.* Ph.D. Dissertation. University of Padova.

[21] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208.

[22] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.* 41 (1992), 1054–1068. Issue 9.

[23] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28, 1 (2004), 7–26.

[24] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. 1992. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Trans. Comput.* 41 (1992).

[25] Eric Torng. 1998. A Unified Analysis of Paging and Caching. *Algorithmica* 20, 2 (1998), 175–200.