Tight Bounds for Parallel Paging and Green Paging

Kunal Agrawal* Michael A. Bender† Rathish Das† William Kuszmaul‡
Enoch Peserico§ Michele Scquizzato§

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Abstract

In the *parallel paging* problem, there are p processors that share a cache of size k. The goal is to partition the cache among the processors over time in order to minimize their average completion time. For this long-standing open problem, we give tight upper and lower bounds of $\Theta(\log p)$ on the competitive ratio with $O(1)$ resource augmentation.

A key idea in both our algorithms and lower bounds is to relate the problem of parallel paging to the seemingly unrelated problem of *green paging*. In green paging, there is an energy-optimized processor that can temporarily turn off one or more of its cache banks (thereby reducing power consumption), so that the cache size varies between a maximum size k and a minimum size k/p. The goal is to minimize the total energy consumed by the computation, which is proportional to the integral of the cache size over time.

We show that any efficient solution to green paging can be converted into an efficient solution to parallel paging, and that any lower bound for green paging can be converted into a lower bound for parallel paging, in both cases in a black-box fashion. We then show that, with $O(1)$ resource augmentation, the optimal competitive ratio for deterministic online green paging is $\Theta(\log p)$, which, in turn, implies the same bounds for deterministic online parallel paging.

1 Introduction

The problem of managing the contents of a ***cache*** (i.e., a small ***fast memory***) is critical to achieving good performance on large machines with multi-level memory hierarchies. This problem is classically known as ***paging*** or ***caching*** [6]. When a processor accesses a location in fast memory, the access cost is small (the access is a ***hit***); when it accesses a location that is not in fast memory, the access cost is large (the access is a ***miss*** or a ***fault***). The paging algorithm decides which ***pages*** (or ***blocks***) remain in fast memory at any point in time or, in other words, which page(s) to evict when a new page is brought into fast memory. This problem is generally formulated as being ***online***, meaning that the paging algorithm does not know the future requests.

Sequential paging—when there is a single processor accessing the fast memory—has been studied for decades and is a very well-understood problem [3,6,23,28], including several of its extensions.

In this paper, we study the problem of ***parallel paging*** where p processors share the same fast memory of some size k. Each processor runs its own program, and the set of pages accessed by different programs are disjoint. At each point in time, the paging algorithm gets to decide how much cache goes to each processor, and also gets to dictate each processor's eviction strategy. The goal is to share the small memory among the processors in a way that minimizes some objective function of processors' completion times. We focus on minimizing average completion time, and we also give bounds on makespan (i.e., maximum completion time) and median completion time.

The parallel paging problem introduces complexity that is not seen in the sequential problem. First, multiple processors compete for the same resource and the paging algorithm must decide, for each processor and at each time, how many and which of its pages to keep in cache. The marginal benefit of having more memory may vary across processors and this relationship may not have good structure. For instance, it may be that Processor 1 derives more marginal benefit from one extra page compared to Processor 2 while at the same time, Processor 2 derives more benefit from ten extra pages compared to Processor 1. In addition, this marginal benefit of extra cache can vary over time and the online paging algorithm must change the number of pages of fast memory allocated to the processors accordingly. Complicating matters even further, the actual scheduling of processors matters. For instance, running a small subset of processors—and temporarily stalling all others—may allow for better performance compared to running all processors. Therefore, how the different processor's accesses are *interleaved* is an important part of what the parallel paging algorithm must decide, and different interleavings of accesses can lead to vastly different performances.

Whereas sequential paging has been understood for decades [3, 6, 23, 28], parallel paging has largely resisted analysis. The only known upper bounds for parallel paging [2, 7, 11, 12, 17, 19, 24] consider relaxations of the problem in which the interleaving of accesses by different processors

*Washington University in St. Louis, USA. Email: kunal@wustl.edu.
†Stony Brook University, USA. Email: {bender, radas}@cs.stonybrook.edu.
‡MIT, USA. Email: kuszmaul@mit.edu.
§Università degli Studi di Padova, Italy. Email: enoch@dei.unipd.it, scquizza@math.unipd.it.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

is fixed ahead of time—in particular, this is enforced by making it so that, whenever any processor blocks on a miss, *all of the processors block*, which in turn eliminates a large amount of the parallelism inherent to the problem. The known lower bounds [15, 22], on the other hand, focus only on the offline parallel paging problem, and on analyzing the performance of traditional paging algorithms such as LRU. Parallel paging has also been extensively studied within the systems community, particularly after multicore processors became mainstream—starting from some pioneering work on (offline and online) heuristics that dynamically adjust the sizes of the cache partitions dedicated to each processor (see, e.g., [8, 20, 29–32]).

This Paper. In this paper, we consider this long-standing open problem of parallel paging in its general form when the interleavings are not fixed. As in [15, 22], we analyze this problem in a ***timed*** model where a hit takes unit time and a miss takes s time units for some parameter $s \geq 1$.[1] Note that, in the traditional model used for paging [6, 28], a hit costs 0 and a miss costs 1 — we will call this the 0-1 model. The timed model is more general and captures the 0-1 model as a special case (by letting s tend towards infinity).

An important feature of the timed model is that it captures the passage of time even when a processor experiences a hit. Modeling this passage of time allows for a more fine-grained analysis of parallel paging since it allows us to compare the progress of one processor to another even if one is experiencing hits while the other is experiencing misses. This avoids the unrealistic assumption in the 0-1 model that a particular processor can have an infinite number of hits in the time that another processor has a single miss.

We give tight upper and lower bounds for the deterministic online parallel paging problem in the timed model showing that the optimal competitive ratio for average completion time is $\Theta(\log p)$, using constant resource augmentation. (Resource augmentation is perhaps the most popular refinement of competitive analysis that allows bypassing overly pessimistic worst-case bounds by endowing the online algorithm with more resources than the offline optimum it is compared to [6].) We also consider makespan, proving a lower bound of $\Omega(\log p)$ and an upper bound of $O(\log^2 p)$ for the competitive ratio. This result represents significant progress on a long-standing open problem—it was first articulated by Fiat and Karlin [12] in 1995 and has remained open even in the 0-1 model. A remarkable feature of our algorithms is that they are oblivious to s, achieving optimal competitive ratios for all values of s simultaneously.

A foundational idea in both our algorithms and lower bounds is to relate the problem of parallel paging to the (seemingly unrelated) problem of ***green paging***, which focuses on minimizing the memory usage (hence, e.g., the energy consumption) of a (single) processor's cache for a computation, and is a problem of independent interest.

Green Paging. In green paging, the fast memory consists of ***memory banks*** that can be turned on or off over time. Memory banks that are active (i.e., turned on) can store pages—and thus requests for those pages result in a hit—but also consume energy; memory banks that are inactive cannot store pages, but do not consume energy. The goal in green paging is to minimize the total energy consumption of a computation. More formally, there is a single processor that is running, and the green paging algorithm gets to control both the page-eviction policy and the size of the processor's cache over time, assigning any size between a minimum of k/p and a maximum of k, where p is a given parameter. The goal is to service the request sequence while minimizing the integral of memory capacity over time—a quantity we call ***memory impact***.[2] This is a simple model for studying the total amount of memory usage over time by a computation.

In this paper, green paging serves both as a problem to be studied on its own, and as an *analytical tool* for studying parallel paging. Indeed, both our upper and lower bounds hinge on unexpected relationships between the two problems. We remark 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 when "translating" one model into the other.

Results. We now summarize our results for deterministic online parallel and green paging. All of the results assume a constant factor of resource augmentation (which is necessary even for sequential paging [5, 13, 28]).

- **Relating parallel and green paging.** We show that green and parallel paging are tightly related. In particular, *any* green paging algorithm with k memory can be translated "black box" into a parallel paging algorithm with $O(k)$ memory capacity. If the former is online, so is the latter. If the former has a competitive ratio of β, then the latter achieves an average completion time with a competitive ratio of $O(\beta)$; additionally, the latter achieves a $(1-\varepsilon)$-completion time (i.e., the time to complete all but an ε fraction of sequences) that is $O(\beta \log(\varepsilon^{-1}))$-competitive with the optimal $(1-\varepsilon/2)$-completion time.

[1] We use the letter s to be consistent with the notation used in the ***full access cost model*** [6], which charges 1 for an access to fast memory and $s \geq 1$ to move a page from slow to fast memory.

[2] Note that it is not always optimal to keep the cache size at k/p since this can lead to a larger number of misses, thereby increasing the running time of the computation. As an example, say a processor accesses 4 pages in round-robin, accessing each page t times. If we give the processor a cache of size 4, it will finish in time $4t$ for a total memory impact of $16t$. On the other hand, if we give it a cache of size 2, at least half of its accesses will be misses. Therefore, the running time will be at least $2st$ for a total memory impact of $4st$. For any $s > 4$, allocating a cache of size 4 results in a smaller memory impact than allocating a smaller cache of size 2.

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

We also prove a relationship between lower bounds for the two problems. If we have an online lower bound construction against online algorithms for green paging, achieving a competitive ratio of $\Omega(\beta)$, then all online algorithms for parallel paging must also have competitive ratio $\Omega(\beta)$ for both mean completion time and maximum completion time.

- **Tight bounds for green paging.** For a lower bound, we show that any online deterministic algorithm has a competitive ratio of at least $\Omega(\log p)$ even with $O(1)$ resource augmentation. And for an upper bound, we give a simple memory allocation algorithm that is both online and *memoryless* (i.e., it does not depend on future or past requests) and that can be combined with LRU replacement to achieve the optimal competitive ratio of $O(\log p)$.
- **Tight bounds for parallel paging.** Using the previous two results, we obtain tight upper and lower bounds of $\Theta(\log p)$ for the parallel paging problem, both when the objective is to minimize mean completion time, and when the objective is to minimize the time spent completing a constant fraction of processes. We also arrive at an upper bound of $O(\log^2 p)$ and a lower bound of $\Omega(\log p)$ for the competitive ratio of optimizing makespan.

Bottom line: optimize memory impact. The relationship between green and parallel paging is powerful. For decades, little progress has been made on parallel paging partly because it has been unclear how to handle the interleaving and interference between different processors. The algorithms in this paper give a clear lesson: rather than focusing on the interactions between processors, and rather than greedily focusing on optimizing the *running times* of processors, one should instead focus on minimizing the *memory impact* of each processor (the same value that is optimized by green paging!). This—along with basic load balancing to keep processors from getting too far ahead or behind—allows the processors to share the cache with each other in the most constructive possible way.

Related Work on Sequential Paging. As mentioned above, sequential paging has been studied for decades for a two-layer memory heirarchy in the 0-1 model. The simple algorithm LFD (Longest Forward Distance) that evicts the page accessed furthest in the future has long been known to be optimal [3, 23]. In the *online* setting, where the algorithm does not know the future, the *competitive analysis* framework is typically used to analyze algorithms. In the 0-1 model, an online paging algorithm has a ***competitive ratio*** of (no more than) β if, for every request sequence, it incurs at most β times as many faults as an optimal offline algorithm incurs with a memory of capacity $h \leq k$ (plus an additive constant independent of sequence length). The ratio $\alpha = k/h$ is called the ***resource augmentation*** factor. Many simple, deterministic algorithms including LRU, FIFO, FWF, and CLOCK have a competitive ratio of $\frac{k}{k-h+1}$ [6, 28]; and the same ratio holds for RAND [6]. This ratio is optimal for deterministic algorithms, and even for randomized ones if page requests can depend on previous choices of the paging algorithm. Since $\frac{k}{k-k/2+1} < 2$, this ratio implies that these algorithms never fare worse than the optimal offline algorithm would on a memory system with half the capacity and twice the access cost.

Related Work on Parallel Paging. The theoretical understanding of the parallel paging problem remains incomplete. Most prior positive results assume that the p request sequences are combined into a single fixed interleaved sequence, to be serviced by selecting which pages to keep in memory so as to minimize the total number of faults [2, 7, 12, 17, 19] or other metrics [24]. That is, the speed at which processors progress relatively to each other is treated as being *independent* of the number of page faults made by each processor, even though in reality a processor that incurs few faults will progress much faster than one that incurs many faults. Feuerstein and Strejilevich de Loma [11] further relax the problem so that they can choose the interleaved sequence rather than assuming that it is given.

An unfortunate consequence of the fixed-interleaving assumption is that whenever a processor incurs a fault, all other processes "freeze" until the fault is resolved. This negates much of the inherent parallelism in the problem since, when a processor encounters a fault, other processors should continue working. However, when one doesn't make this assumption, the problem becomes much more complicated since processors can advance while other processors are blocking on faults, and thus the relative rates at which processors advance is determined by when they hit or miss. This means that the *actual interleaving of request sequences of different processors depends on the paging algorithm*, since the paging algorithm determines when processors hit or miss.

Some recent works [10, 15, 16, 22] do not make the fixed-interleaving assumption; these works investigate the complexity of the offline problem and show lower bounds for traditional paging algorithms such as LRU, or consider restricted models. However, no general upper bounds or lower bounds are known, and the fully general problem of how to manage a shared fast memory among multiple processors has remained open.

Related Work on Green Paging. The last decade has seen a surge in interest for paging models where memory capacity is not static, but can instead change over time [2, 4, 5, 9, 14, 21, 25, 26]. One justification for such models is the increased popularity of virtualization/cloud services: the amount of physical memory allotted to a specific virtual machine often varies considerably over time based on the number and priority of other virtual machines supported by the same

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

hardware. Another justification for dynamic capacity models lies in the ability of modern hardware to turn off portions of memory so as to reduce power, often with the goal of minimizing the overall energy used in a computation—this is the task we refer to as green paging.

The first work to address green paging was [9], 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 of the average capacity over all requests. This problem has been investigated by López-Ortiz and Salinger [21] and later, in the more general version where pages have sizes and weights, by Gupta et al. [14]. Subsequent work, and in particular the ***elastic paging*** of [25, 26], showed 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 are optimally competitive, with a competitive ratio that is extremely close albeit not quite equal to the classic $k/(k-h+1)$. A similar line was taken by ***adaptive caching*** [5] with a slightly different cost model. These results [5, 25, 26] imply that green paging is a problem of memory allocation: once memory is allocated, one can simply use LRU for page replacement—as its cost will be within a factor $O(1)$ of the optimal (for that memory allocation).

We remark that portions of this work are based on preliminary results contained in the thesis [27]. A preliminary version of this work was presented as a brief announcement at SPAA 2020 [1].

2 Technical Overview

This section gives an overview of our main results and of the techniques that we use to prove them. Recall that we will use the timed model where cache hits take time 1, and cache faults take time s for some integer $s \geq 1$. A detailed specification of the models for parallel paging and for green paging can be found in Section 3.

The relationship between Parallel and Green Paging. In parallel paging, the fact that processors must share a cache of size k suggests that the cache-usage per processor should be treated as a critical resource. Green paging, in turn, optimizes this resource for an *individual* processor by minimizing the total memory impact for that processor.

Green paging does not concern itself with minimizing running time directly though—for example, a green paging algorithm might choose to use a very small portion of the cache for a long time rather than a larger portion for a short time. Additionally, since green paging focuses on only a single processor, it does not say anything about the interactions between concurrent processors. These interactions (i.e., the ways in which the working sets for different processors change relative to each other over time) play a critical role in the parallel paging problem.

One of the key contributions of this paper is that, in spite of these obstacles, memory impact really is the right way to think about parallel paging. Even though parallel paging involves complicated interactions between processors, we show that the problem can be decomposed in a way so that each individual processor can be optimized separately. The result is a black-box reduction from the parallel-paging problem to green paging. Remarkably, the opposite direction is also true: any online lower-bound construction for green paging can be transformed in a black-box fashion to obtain an online lower-bound construction for parallel paging.

By proving a tight relationship between green and parallel paging, and then giving tight bounds for green paging, we immediately obtain tight bounds for parallel paging as a result.

In the rest of this section, we first describe a series of simplifications that allow us to think about each individual processor's use of cache in terms of a so-called boxed memory profile. We then explain how to achieve tight bounds for green and parallel paging.

2.1 A Useful Tool: Box Profiles

In the green paging problem, the paging algorithm sets a ***memory profile*** $m(i)$, which dictates how much cache the processor uses at each point in time. A key insight, however, is that we need only consider profiles with a certain nice geometric structure, called box profiles.

Box profiles. A ***memory box of height h*** is a time-interval of length $\Theta(sh)$ during which a processor is allocated exactly h memory. We call a memory profile m a ***box profile*** if it can be decomposed into a sequence of memory boxes $b_1, b_2, \ldots$.

Box profiles are without-loss-of-generality in the following sense: If an online algorithm for green paging produces a memory profile m, then the algorithm can be modified (online) to instead produce a *box profile* m'. Moreover, the box profile m' will incur at most a constant-factor more memory impact than does m.

The intuition behind this transformation is the following: without loss of generality, the profile m never grows at a rate of more than 1 per s time steps, because fetching a page from the slow memory to the fast memory takes s time (although it can shrink arbitrarily fast). Thus, whenever the memory profile m is at some height h, the profile must have already been at height $\Omega(h)$ for time at least $\Omega(sh)$. This naturally allows for one to decompose the profile into (overlapping) chunks where each chunk closely resembles a box. Making these chunks not overlap, and so that the decomposition is online, requires several additional ideas that we give in Section 4.

Box profiles were previously used as analytical tools for understanding cache-adaptive algorithms [4, 5], which

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

are algorithms that exhibit optimal cache behavior in the presence of a varying-size cache. An interesting feature of our work is that box profiles play an important role not just analytically, but also *algorithmically* in our treatments of green and parallel paging.

Simplifying box profiles: compartmentalization, smooth-growth, and optimal eviction. We can also assume that the box profile m' has additional nice properties, the simplest of which are that every box has a power-of-two height 2^j, and that the height-to-width ratio of every box is always the same.

On top of these, we can assume ***compartmentalization***. This property says that, at the beginning of each box b's lifetime, the cache is flushed (i.e., the cache size briefly dips to 0). This means that each box of height h must incur h cache misses *just to populate its cache*. These cache misses can be handled by increasing the box's width by an additional sh. Since the box already had width $\Theta(sh)$, the increase does not change the asymptotic memory impact of the box. Compartmentalization plays an important role in the design of algorithms for parallel paging, since it means that consecutive boxes in a processor's profile need not be scheduled adjacently in time.

We can also assume the ***smooth-growth property***: whenever two boxes b_i, b_{i+1} come after one another, the height of the latter is at most twice that of the former. This property will be especially useful when proving lower bounds.

Finally, because each box in a box profile has a fixed height, LRU in a box of height 2^j is guaranteed to perform 2-competitively with the optimal eviction policy OPT in a box of height 2^{j-1} [28]. Up to a constant factor of resource augmentation, we can therefore assume without loss of generality that the optimal policy OPT is used within each box.

2.2 Tight Bounds for Green Paging

Consider a green paging instance with maximum memory k and minimum memory k/p. One can assume without loss of generality that k, p are powers of 2. In addition, we can assume that the page replacement strategy is optimal (or LRU) within each box—therefore, the algorithm needs only decide the sequence of boxes to be used. Furthermore, as discussed at the beginning of the section, one can also assume that the asymptotically optimal solution OPT is a box profile using boxes with heights $k/p, 2k/p, 4k/p, \ldots, k$.

A Universal Box Profile. In Section 6.2, we present the ***BLIND*** algorithm for green paging (specifically, to decide the sequence of boxes that the algorithm should use), which achieves competitive ratio $O(\log p)$ using constant resource augmentation. A remarkable property of this algorithm is that the sequence of boxes that it uses is *oblivious* to the input request sequence σ. In particular, the BLIND algorithm always uses a fixed sequence of boxes that we call the ***universal box profile*** U.

We construct the universal box profile U by performing repeated traversals of a tree $\mathcal{T}$, where each node in $\mathcal{T}$ is associated with a certain box size. The tree $\mathcal{T}$ has $1 + \log_2 p$ levels, and each internal node in the tree has four children. For each of the levels $0, 1, 2, \ldots, \log_2 p$, starting at the leaves, the nodes in level i are boxes of height $2^i k/p$. In particular, the root node is a box of height k, and the leaves are boxes of height k/p. The key property of this tree is that each internal node's memory impact is equal to the sum of the memory impact of all its children and therefore, the sum of the *memory impacts* of the boxes at each level i is the same for every level.

The universal box profile U is constructed by performing a postorder traversal of the tree $\mathcal{T}$ (i.e., we start in the bottom left leaf, and we always visit children before visiting parents). Whenever the postorder traversal completes, it then restarts.

Analyzing the BLIND Algorithm. In Section 6.2, we prove the following theorem.

Theorem 1. *Using resource augmentation $\alpha = 2$, the competitive ratio of BLIND is $O(\log p)$.*

To analyze the BLIND algorithm, consider the optimal box profile OPT, which uses boxes $x_1, x_2, \ldots$, and compare it to the universal box profile U, which uses boxes $y_1, y_2, \ldots$. Let $U_{\text{prefix}} = \langle y_1, y_2, \ldots, y_j \rangle$ be the smallest prefix of U that contains OPT as a subsequence, and call a box y_i ***successfully utilized*** if it is used in the OPT subsequence.

The challenge is to bound the total memory impact of U_{prefix} by $O(\log p)$ times the total memory impact of OPT. In the rest of this overview, let us focus on only the first tree-traversal in U_{prefix}.

The key combinatorial property of the BLIND algorithm is that, for every root-to-leaf path P in the tree $\mathcal{T}$, at least one box in that path P is guaranteed to be successfully utilized. In particular, in Section 6.2 we show that if the path P has nodes $p_1, p_2, \ldots, p_j$ where p_j is a leaf, then once BLIND's postorder traversal reaches p_j, the next box that the algorithm successfully utilizes is guaranteed to be one of $p_1, p_2, \ldots, p_j$.

The tree $\mathcal{T}$ is designed so that each box b has exactly the same memory impact as the sum of its descendant leaves. Since every root-to-leaf path contains at least one successfully utilized box, it follows that: the sum of the memory impacts of successfully utilized boxes is at least as large as the sum of memory impacts of *all* leaves.

By design, the sum of the memory impacts of the leaves in $\mathcal{T}$ is $1/(1 + \log p)$ of the total memory impact of all boxes in $\mathcal{T}$. The consequence is that, the memory impacts of successfully utilized boxes must represent at least a $1/(1 +$

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

$\log p)$ fraction of U_{prefix}'s memory impacts, as desired.

It is interesting that the BLIND algorithm achieves a competitive ratio of $O(\log p)$ while being oblivious to the input sequence σ. On the other hand, the fact that BLIND is oblivious gives hope that an even smaller competitive ratio might be achievable by an adaptive algorithm. Remarkably, this turns out not to be the case.

Lower-Bound Construction: Go Against the Flow. In Section 6.1, we prove the following:

Theorem 2. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Let ALG be any deterministic online algorithm for green paging, and let α be the amount of resource augmentation. Then, the competitive ratio of ALG is $\Omega\left(\frac{\log p}{\alpha}\right)$.*

For simplicity, here we focus on the case where the resource augmentation $\alpha = 1$, where $s \geq p$, and where the minimum box-height for OPT is normalized to 1 (meaning that $k = p$).

Consider an request sequence σ for ALG in which, at every step we request the element i most-recently evicted from ALG's cache. (In particular, ALG misses on every request.) Let c_{ALG} be the total memory impact of ALG on the sequence σ.

We now design (offline based on σ) an algorithm OFF that achieves total memory impact $c_{\text{OFF}} \leq O(c_{\text{ALG}}/\log p)$.

The algorithm OFF selects a threshold 2^j and always does the *opposite* of what ALG does with respect to that threshold. Namely, whenever ALG has cache-size 2^j or greater (we call these time intervals ***islands***), OFF sets its cache-size to 1. And whenever ALG has cache-size less than 2^j (we call these time intervals ***seas***), OFF sets its cache-size to 2^j.

For now, the only constraint that we will place on the threshold 2^j is that $\log p \leq 2^j \leq k/\log p$. Later, however, we will see that a careful selection of 2^j is essential to complete the proof.

Analyzing the Lower-Bound Construction. In order to analyze c_{OFF}, we begin by considering the islands. During these time intervals, both ALG and OFF miss on every request, but ALG uses a cache of size $2^j \geq \log p$ whereas OFF uses a cache of size 1. Thus the memory impact of ALG is at least a factor of $\log p$ larger than that of OFF during the islands.

Next, we consider the seas. Each of these time intervals has a ***transition cost*** for OFF, in which OFF must transition from a cache of size 1 to a cache of size 2^j, and incurs $2^j - 1$ misses in order to fill its cache. If we ignore the transition costs for a moment, then it turns out that OFF never incurs any cache misses within a sea. This is because the request sequence σ is designed to only have memory footprint $\leq 2^j$ within a given sea. Since ALG always misses, each request costs ALG at least s. On the other hand, since OFF never misses (but uses a cache of size 2^j), each request costs OFF $2^j \leq k/\log p \leq s/\log p$. Again we have that the memory impact of ALG is at least a factor of $\log p$ larger than that of OFF.

If the threshold size 2^j is not selected carefully, then the transition costs can end up dominating the other costs in OFF. Indeed, if not for the transition costs, one could select the threshold 2^j to be $\sqrt{p}$ and force a competitive ratio of $\Omega(\sqrt{p})$ (rather than $\Omega(\log p)$).

In order to minimize the transition costs, one must select the threshold size 2^j in a special way. We select 2^j to be the box-height in the range $[\log p, k/\log p]$ that contributes the least total memory impact to ALG over all such box heights. That is, for $i \in \{0, \ldots, \log k\}$ define $S(2^i)$ to be the total memory impact incurred by boxes of height 2^i in ALG, and define $j = \arg\min_{2^i=\log p}^{k/\log p} S(2^i)$. Since $\sum_i S(2^i) = c_{\text{ALG}}$, one can deduce that $S(2^j) \leq O(c_{\text{ALG}}/\log p)$.

To complete the proof, we show that the sum of the transition costs is $O(S(2^j))$. By the smooth-growth property, between every sea and island, ALG always has at least one box of height 2^j. The cost of this box for ALG is within a constant factor of the corresponding transition cost for OPT. This establishes that the total of the transition costs is $O(S(2^j)) \leq c_{\text{ALG}}/\log p$, as desired.

2.3 Using Green Paging to Solve Parallel Paging

In Section 5.1, we prove the following theorem:

Theorem 3. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging with competitive ratio $O(\beta)$ for average completion time. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

We also extend our analysis to show that the same algorithm achieves guarantees on the completion time for a *given number* of processors. This provides a continuous tradeoff between average-completion-time type guarantees and makespan-type guarantees, and allows for us to obtain guarantees for metrics such as *median* completion time.

Theorem 4. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging that achieves the following guarantee: For any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction of 2^{-i-1}. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

Two important special cases of Theorem 4 are: the

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

makespan is within a factor of $O(\beta \log p)$ of optimal, and the median completion is within a factor of $O(\beta)$ of the optimal time to complete at least $3/4$ of the processors.

In the rest of this section, we give an overview of the proof of Theorem 3, meaning that our focus is on minimizing average completion time. For ease of notation, we will do the reduction in the special case where $\beta = 1$ (meaning we are given an optimal algorithm for green paging) and $\alpha = \Theta(1)$. In the following discussion, let OPT denote the optimal solution to the parallel paging problem (for minimizing average-completion time).

A Warmup: The Box-Packing Algorithm. We begin by describing a simple parallel-paging algorithm which we call the Box-Packing algorithm. The Box-Packing algorithm behaves well on a certain special class of inputs (which we call uniform-impact), but will not guarantee small average completion times in general. Later in the section, we will use the algorithm as a building block to construct a different algorithm that does offer general guarantees.

During the Box-Packing algorithm, each processor runs an instance of green paging in order to produce a sequence of boxes with heights between k/p and $k/2$. The Box-Packing algorithm then greedily packs these boxes into a memory of size k over time using the following approach: if at any point in time, less than $k/2$ of the memory is being used, then the Box-Packing algorithm selects a processor q that is not currently executing (if such a processor exists), and places the next box for that processor into the cache.

An important feature of the Box-Packing algorithm is that, when picking which processor q to allocate space for in the cache, the algorithm performs a form of load balancing. Rather than giving priority to the processors q that have run for the least total time (which might seem natural), the algorithm instead selects the processor q that has incurred the *smallest total memory impact* so far, out of the processors that are idle. We call this the ***impact-balancing property***.

When the Box-Packing Algorithm Does Well: Uniform-Impact Processors. Despite not being an optimal algorithm for parallel paging in general, the Box-Packing algorithm does do well in one important special case: the case where the processors are ***uniform-impact***.

For each processor q, let I_q denote the total memory impact used by the green-paging solution for q. We call the processors $1, 2, \ldots, p$ ***uniform-impact*** if $I_1 = I_2 = \cdots = I_p = I$ for some I.

The fact that the processors are uniform-impact ensures that the cache is always close to fully utilized. In particular, a critical failure mode for the Box-Packing algorithm is if the size-k memory is under-utilized (i.e., less than $k/2$ of the memory is allotted to boxes), but there are no remaining processors to schedule (because too many processors have already finished). If the processors are uniform-impact, however, then the Box-Packing algorithm finishes all of the processors at roughly the same time, avoiding the under-utilization failure mode.

Because the memory is close to fully utilized, the total running time is equal to the total memory impact of all processors divided by k, i.e., $\Theta(pI/k)$.

On the other hand, assuming that the processors are impact-balanced, we can show that the optimal average completion time is also $\Theta(pI/k)$. In particular, in OPT, the $p/2$ processors that finish first must together incur total memory impact at least $\Omega(pI)$,[3] thereby requiring time at least $\Omega(pI/k)$. Thus the $p/2$ processors that finish last in OPT each incur running times $\Omega(pI/k)$.

The Final Algorithm: The Phased Box-Packing Algorithm. In order to do well when the processors are not uniform-impact, we introduce the Phased Box-Packing algorithm. This is the algorithm that gives the guarantees in Theorems 3 and 4.

The algorithm consists of $1 + \log p$ phases $0, 1, 2, \ldots, \log p$, where phase i begins at the first point in time where only $p/2^i$ processors remain. During each phase i, the Phased Box-Packing algorithm runs an instance of the Box-Packing algorithm on the $p/2^i$ processors that remain, and then terminates that instance prematurely at the end of the phase. Let ΔT_i be the running time of the i-th phase, and let S_i denote the set of processors that finish during the i-th phase.

Analyzing the Phased Box-Packing Algorithm by Comparing Phases. The key to performing a competitive analysis of the algorithm is to analyze each phase based not on the average completion time of the processors S_i that finish in *that* phase, but instead based on the average completion time of the processors S_{i+1} that finish in the next phase. Note that the processors S_{i+1} represent a $1/4$ fraction of the processors that execute during phase i. By the impact-balancing property, it follows that the processors S_{i+1} incur a constant fraction of the memory impact that is incurred in phase i, and that all of the processors in S_{i+1} incur (almost) the same memory impacts as one-another in phase i. This means that, by ignoring the other processors that execute during phase i, one can treat the processors in S_{i+1} as being uniform-impact, and conclude that the average completion time in OPT for the processors in S_{i+1} is $\Omega(\Delta T_i)$.

Phase i contributes running time at most ΔT_i to at most

[3]Without loss of generality, we assume that OPT always allocates at least $\Omega(k/p)$ space to each processor, since up to a factor of 2 in resource augmentation we can feel free to spread half of OPT's memory equally among the processors and limit OPT's control to the other half. Note that, without this minimum-allocation height assumption, the following problem might arise: OPT could use boxes of height $o(k/p)$, possibly achieving memory impact less than I for some processor. The minimum-allocation height assumption fixes this by ensuring that all of OPT's boxes have height $\Omega(k/p)$.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

$2|S_{i+1}|$ processors. On the other hand, the processors S_{i+1} require average running time at least $\Omega(\Delta T_i)$. Thus we can perform a charging argument where we use the running time of processors S_{i+1} in OPT to pay for the running times of all processors in phase i. This proves that the algorithm is $O(1)$-competitive with OPT.

2.4 Transforming Green Paging Lower Bounds into Parallel Paging Lower Bounds We now consider how to transform an arbitrary lower-bound construction for green paging into a matching lower-bound construction for parallel paging. Section 5.2 proves the following theorem.

Theorem 5. *Suppose there exists a green paging lower bound construction $\mathcal{L}$ that achieves competitive ratio $\Omega(\beta)$. Then all deterministic parallel paging algorithms (that use $\alpha \leq O(1)$ resource augmentation) must incur competitive ratio $\Omega(\beta)$ for both average-completion time and makespan.*

Consider a deterministic parallel paging algorithm A that has resource augmentation $\alpha = \Theta(1)$. We can assume without loss of generality that A always allocates space at least $k/(2p)$ to every processor. In particular, these minimum allocations combine to only use half of the memory, which up to a constant factor in resource augmentation can be ignored.

As A executes the p processors on their request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ (which we will define in a moment), each processor's request sequence σ_i is executed with some memory profile m_i. Since m_i always allocates between $k/(2p)$ and k memory, one can think of m_i as being a green-paging solution for sequence σ_i.

To construct adversarial sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ for A, we use the lower-bound construction $\mathcal{L}$ to construct each of the sequences in parallel. We terminate each of the sequences σ_i once the corresponding memory profile m_i produced by A reaches some large memory impact R.

The fact that each of the profiles m_i have the same memory impacts R allows for us to lower bound the average completion time for algorithm A. In particular, the first $p/2$ processors to complete must incur total memory impact at least $\Omega(pR)$, thereby incurring total running time at least $\Omega(pR/k)$. It follows that the final $p/2$ processors to complete each take time more than $\Omega(pR/k)$. Thus $\Omega(pR/k)$ is a lower bound for both the average completion time and the makespan of A.

In order to complete the proof, we construct an alternative parallel-paging solution B that has makespan (and thus also average completion time) only $O(pR/k\beta)$. Note that, because algorithm A is analyzed with resource augmentation α, the maximum memory size for B is k/α.

Now we consider the optimal green-paging solution for each request sequence σ_i, where the optimal solution is restricted to have minimum box height $k/(4\alpha p)$ and maximum box height $k/(2\alpha)$ (i.e., the optimal solution is limited by a factor-of-2α resource augmentation in comparison to the solutions produced by A). Let m_i^{OPT} be the (boxed) memory profile produced by the optimal green-paging solution for σ_i. By the definition of the lower-bound construction $\mathcal{L}$, we know that the memory impact of each m_i^{OPT} is only $O(R/\beta)$.

To construct the parallel-paging solution B, we simply perform the Box-Packing algorithm from Section 2.3 on the box profiles $m_1^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$. In particular, whenever the total memory allocated to processors is less than $\frac{k}{2\alpha}$, algorithm B selects a processor i out of those not currently executing (if there is one) and allocates space for the next box the profile m_i^{OPT}. Note that the box is guaranteed to fit into B's memory of size k/α, since the maximum box height in any profile m_i^{OPT} is only $k/(2\alpha)$.

A simple way to analyze the average running time of B is to note that (without loss of generality) the request sequences $m_1^{\text{OPT}}, m_2^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$ are impact balanced, each having the same memory impact $I = \Theta(R/\beta)$. By the analysis in Section 2.3, the total makespan for B is only $O(pI/k) = O(Rp/k\beta)$. Since this is a factor of $\Omega(\beta)$ smaller than the average completion time and makespan of A, this completes the lower-bound transformation.

2.5 Putting Pieces Together Combining the upper and lower bounds for green paging in Section 2.2, we arrive at the following tight bound on green paging.

Theorem 6. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Then there exists a deterministic online algorithm (with $O(1)$ resource augmentation) that achieves competitive ratio $O(\log p)$. Moreover, this competitive ratio is asymptotically optimal for deterministic online algorithms.*

Note that the assumption in Theorem 6 that $s \geq p^{1/c}$ is natural for the following reason: if any green paging algorithm ever uses a square of height more than sk/p, then the algorithm would be better off using a square of height k/p (and just incurring cache misses everywhere). Thus the natural parameter regime for green paging is when the maximum box-height k is less than sk/p, meaning that $p \leq s$.

By combining the upper and lower bounds for green paging with the reductions in Sections 2.3 and 2.4, we arrive at the following bounds for parallel paging:

Theorem 7. *There exists an online deterministic algorithm for parallel paging that achieves competitive ratio $O(\log p)$ for average completion time, using resource augmentation $O(1)$. Moreover, for any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i \log p)$ of the optimal time to complete all but a*

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

fraction of 2^{-i-1}. One consequence of this is that a competitive ratio of $O(\log^2 p)$ is achieved for makespan. Furthermore, any deterministic parallel paging algorithm must have competitive ratio $\Omega(\log p)$ for both average completion time and makespan, as long as $s \geq p^{1/c}$ for some constant c.

3 The Models

Before proceeding with our results, we first take a moment to discuss the models for green paging and parallel paging in detail and, in particular, to highlight some of the unintuitive differences between these problems and classic paging.

3.1 The Green Paging Model *Green paging* models computations in environments, such as cloud computing or low-power computing, where the amount of memory resources allocated to a given task can be changed dynamically, and the objective is to complete the task minimizing the total amount of memory resources consumed over time.

Formally, in green paging, like in standard paging, an algorithm controls a memory system with two layers: a fast memory that can hold a limited number of pages, and a slow memory of infinite capacity. Accessing a page takes one time step if that page is in fast memory. If a requested page is not currently in fast memory, it can be accessed only after copying it into fast memory; fetching a page from slow to fast memory takes $s \geq 1$ time steps. During the s time steps of a fault, the corresponding fast memory page must be available and not otherwise in use. In practice, $s \gg 1$; we assume for simplicity that s is an integer, allowing us to work with discrete time steps. Pages can be brought into fast memory only when requested, but can be discarded at any time, instantaneously and at no cost. We denote by $P_{ALG}(i)$ the set of pages kept in (or being loaded into) fast memory throughout the i-th time step by a paging algorithm ALG.

Two main differences set green paging apart from classic paging. The first is that the fast memory capacity is not fixed, but in general varies over time *under the control of the paging algorithm*, between a maximum of k and a minimum of k/p pages. We denote by $m_{ALG}(i) \geq |P_{ALG}(i)|$ the fast memory capacity set by ALG throughout the i-th time step. We call the function $m(i)$ the ***memory profile***.[4] The second difference is that the paging algorithm must access any given sequence of page requests $\sigma = \langle r_1, \ldots, r_n \rangle$ minimizing not the total time taken $T_{ALG}(\sigma)$ (or, equivalently, the number of faults) but instead the integral, over that time interval, of the fast memory capacity, that is, $\sum_{i=0}^{T_{ALG}(\sigma)} m_{ALG}(i)$. We call this quantity the ***memory impact*** of a paging algorithm. As mentioned in Section 1, lower capacity does not necessarily translate into a lower memory impact, if it means more page faults and thus more processing time—think of a cycle over four pages serviced with memory capacity $m(i) = 4$ or with memory capacity $m(i) = 2$ for all i.

Solely to simplify the analysis, we also introduce a third, minor, difference: we allow the possibility of idling on any given time step following a page access (or another idle time step). The time step is counted towards the memory integral, but the request sequence does not advance and the memory contents do not change (unless $|P_{ALG}(\cdot)|$ must decrease as a consequence of a reduction of $m_{ALG}(\cdot)$). This is justified by the availability of the No OPeration (NOP) instruction in most processors.

Denote by $r_{ALG}(i)$ the request from σ serviced at time step i by a green paging algorithm ALG. Then ALG is *online* if it determines $m_{ALG}(i)$ and $P_{ALG}(i)$ based solely on $r_{ALG}(1), \ldots, r_{ALG}(i)$. Informally, we define the ***competitive ratio*** with $\alpha \geq 1$ ***resource augmentation*** by comparing the memory impact of the online algorithm with that of an optimal offline algorithm that runs on a system with α times less capacity and pays α times as much for the same capacity, i.e., one whose memory capacity during the i-th time step $m_{OPT}(i)$ lies between $\lfloor k/p\alpha \rfloor$ and $\lfloor k/\alpha \rfloor$, and that incurs a cost equal to $\sum_{i=0}^{T_{OPT}(\sigma)} \alpha \cdot m_{OPT}(i)$. In other words, the memory impact of the optimal algorithm is scaled by a factor α, so that for all j, it costs the optimal algorithm the same amount to allocate j/α memory as it does for the online algorithm to allocate j memory (i.e., allocating a *p-fraction* of memory costs the same for both algorithms). We remark that the main focus of this paper is the case of $\alpha = \Theta(1)$, however, in which case this distinction is not important.

Note that in general we do not assume $k/p = 1$. This models a number of situations of practical interest. In some systems, memory can be only allocated in units significantly larger than a single block. More typically, computing systems incur other running costs (e.g., the power consumed by a motherboard or processor) that cannot be reduced below a certain threshold per unit of time if a computation is running at all, regardless of how little memory is used; in these cases k/p represents the memory capacity below which these other costs become dominant, and thus below which memory capacity reductions cannot grant significant cost reductions, per unit of time, to the system as a whole.

3.2 The Parallel Paging Model *Parallel paging* models computations where multiple processing units, each operating concurrently and independently, share nonetheless the same fast memory—a situation that multicore processors have made over the last two decades the standard on virtually all computing systems from supercomputers to mobile phones. As in classic paging, the goal is to choose at each

[4] This should not be confused with the memory profile function as defined in [5]. In both cases a memory profile function specifies the quantity of memory available at any given time, but (1) in [5], this quantity is set adversarially and not under the control of the algorithm, and (2) in [5] time advances with the page faults of the algorithm.

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

point in time which pages to keep in fast memory so as to minimize some objective function of processors' completion times.

The model we adopt is essentially the one introduced by López-Ortiz and Salinger [22]. We have p processors that share the same fast memory of size k. We assume $k \geq p$. Each processor issues a sequence of page requests; hence we have p sequences of page requests $\sigma_1 = \langle r_1^1, \ldots, r_{n^1}^1 \rangle, \ldots, \sigma_p = \langle r_1^p, \ldots, r_{n^p}^p \rangle$. (Note that the number of processors is denoted by the same parameter, p, that denotes the ratio between the maximum and the minimum memory capacity in the green paging problem. This is intentional since, although the two quantities are drastically different, when showing the equivalence between green paging and parallel paging we will show that they play exactly the same role.) We assume that the sets of pages requested by different processors are disjoint, corresponding to separate processes.

Accessing a page takes one time step if that page is in fast memory (i.e., cache). If a requested page is not currently in fast memory, it can be accessed only after copying it into fast memory; fetching a page from slow to fast memory takes $s \geq 1$ time steps. During the s time steps of a fault, the corresponding fast memory page must be available and not otherwise in use. In practice, $s \gg 1$; we assume for simplicity that s is an integer, allowing us to work with discrete time steps. The goal of the paging algorithm is to choose which pages to maintain in fast memory so as to minimize the *maximum*, *average* or *median* time to service $\sigma_1, \ldots, \sigma_p$. As in green paging, we allow the possibility of idling on any given time step following a page access (or another idle time step).

The parallel paging model allows for all of the processors to make progress in parallel. That is, multiple processors can experience cache misses and cache hits concurrently. However, it may sometimes be useful for the algorithm to temporarily halt some subset of processors (i.e., those processors are simply idle) in order to make the best possible use of the fast memory (e.g., in order to fit the entire working set for some subset of processors into fast memory).

Note that the only interaction between the p processors lies in the fact that the fast memory must be partitioned (dynamically) between them; in particular, we denote by $m_{ALG}^j(i)$ the amount of fast memory allocated by a paging algorithm ALG throughout the i-th time step to the j-th processor (so that $\sum_{j=1}^p m_{ALG}^j(i) \leq k$ for all i) and by $P_{ALG}^j(i)$ the set of its pages in, or being loaded into, the fast memory at that time step. In general, note that $m_{ALG}^j(i)$ *can* be 0 if and only if the j-th processor is idle on the i-th time step. We also remark that, although the order in which each processor accesses its own page sequence is fixed, the sequences will be interleaved in different ways depending on how much memory and thus "speed" each request sequence is allocated.

Denote by $r_{ALG}^j(i)$ the request from σ_j serviced on time step i by a parallel paging algorithm ALG. Then ALG is *online* if $m_{ALG}^1(i), \ldots, m_{ALG}^p(i)$ and $P_{ALG}^1(i), \ldots, P_{ALG}^p(i)$ depend solely on $r_1^1, \ldots, r_{ALG}^1(i), \ldots, r_1^p, \ldots, r_{ALG}^p(i)$. A factor of $\alpha \geq 1$ of *resource augmentation* simply means that the optimal offline algorithm is restricted to a memory of size $\lfloor k/\alpha \rfloor$.

4 A Toolbox for Paging Analysis

The goal of this section is to show how imposing certain constraints on the memory allocation does not significantly degrade performance. In a nutshell, these involve rounding capacity to the next power of two, ensuring that capacity does not change "too often", and periodically lowering capacity to 0. Operating under these constraints significantly simplifies subsequent analyses.

4.1 Memory Expansions Let $m_{ALG}(i)$ denote the memory capacity set by algorithm ALG throughout the i-th clock tick. In our analyses, we shall often compare two algorithms by allotting to the first "more (memory) space" or "more time". A more formal definition of this notion is that of *expansion* of a memory profile function $m(t)$—basically a new memory profile in which each clock tick with memory m is replaced by one *or more clock ticks* with memory *at least* m.

Definition 1. *Consider a memory profile function $m(t)$: $\mathcal{N} \to \mathcal{N}$. An* expansion *of $m(t)$ is pair of functions $\bar{t}(t)$: $\mathcal{N} \to \mathcal{N}$ and $\bar{m}(\bar{t}) : \mathcal{N} \to \mathcal{N}$, with the following properties:*

1. $\bar{t}(t+1) > \bar{t}(t)$.
2. *for each $\bar{t}$, either:*
   (a) *there exists t such that $\bar{t} = \bar{t}(t)$, and then $\bar{m}(\bar{t}) \geq m(t)$, or*
   (b) *there exist t^- and $t^+ = t^- + 1$ such that $\bar{t}(t^-) < \bar{t} < \bar{t}(t^+)$, and then $\bar{m}(\bar{t}) \geq \min(m(t^-), m(t^+))$.*

If $\forall t$ we have that $\bar{t}(t)$ and $\bar{m}(\bar{t}(t-1)+1), \ldots, \bar{m}(\bar{t}(t))$ can be determined solely on the basis of the page requests up to time t, then we say that the expansion is online.

If a paging algorithm can service a request sequence before clocktick T given $m(t)$ memory, then an *optimal offline* algorithm can obviously always service the same request sequence on any expansion of $m(t)$ before (the image of) T – essentially by "wasting" any extra memory, and idling through extra clockticks while preserving the memory contents. Note that it is not true *in general*, for all paging algorithms. Indeed, even if $m(t)$ and $\bar{m}(t)$ are constant, there are online algorithms such as FIFO that service *some* sequences *faster* if given *less* memory (a phenomenon known as Belady's anomaly [6]).

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Definition 2. *The **memory impact** of a memory profile function $m(t)$ is the integral of memory (space) allotted to the profile over time, that is $\sum_t m(t)$.*

For the rest of the paper, we use the memory impact and cost of a memory profile interchangeably.

4.2 Space and Time Normalization To simplify memory management, we consider expansions of memory profiles $m(t)$ with some constraints. More precisely, we show how any "natural" memory profile can be expanded online into another memory profile that satisfies some constraints and incurs a memory impact within a factor of $O(1)$ of the original profile.

With "natural" we mean having a very simple, specific property: on any tick in which allocated memory increases, it increases by at most one page and then remains constant for (at least) $s-1$ more ticks. This reflects the fact that it is pointless to increase the memory profile unless one needs to load one or more pages not currently in memory, which takes s clockticks per page. More formally:

Definition 3. *A memory profile function $m(t)$ is* natural *if, for any t such that $m(t+1) > m(t)$, $m(t+1) = \cdots = m(t+s) = m(t)+1$.*

We remark that the expansions of natural memory profiles we consider are not, in general, natural. The idea is that the expanded profiles are actually wasting space, but their definition makes them easier to handle in proofs. It would not be difficult, albeit extremely cumbersome, to consider natural expansions for which all proofs still work.

The first constraint a memory expansion must satisfy is that the allocated memory should, at any given time, be a power-of-2 multiple of the minimum allowed memory capacity k_{min}.

Definition 4. *A memory profile function $m(t)$ is* space-normalized *if, for all t, there exists some $i \in \mathcal{Z}_0^+$ such that $m(t) = 2^i k_{min}$.*

The second constraint imposes that, roughly speaking, whenever allocated memory changes to a new value m, it should remain m for a number of clockticks that is an integer multiple of $s \cdot m$. This is formally defined as follows.

Definition 5. *A memory profile function $m(t)$ is* time-normalized *if any maximal interval during which $m(\cdot)$ maintains constant value m lasts a number of clockticks equal to an integer multiple of $s \cdot m$. We call such an interval a **box**.*[5]

[5]Recall that in the technical overview (Section 2), for convenience, we defined boxes to always have fixed width $\Theta(sm)$, rather than having width equal to an integer multiple of $s \cdot m$. If one breaks each box (as defined here) into multiple boxes of the form described in Section 2, then one can assume without loss of generality that all boxes have the form described in Section 2.

Figure 1 shows an example of a space- and time-normalized memory profile function. We now show how to

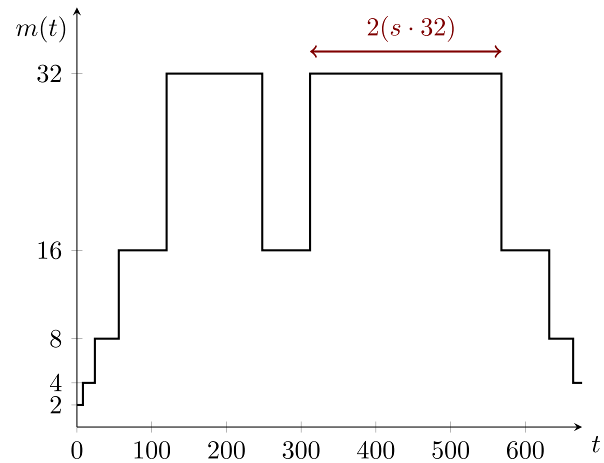

Figure 1: A space- and time-normalized memory profile.

expand online any natural memory profile function $m(t)$ into a space- and time-normalized function $\bar{m}(\bar{t})$ such that the total memory impact incurred by the latter is within a constant factor of that incurred by the former, that is, $\sum_{\bar{t}} \bar{m}(\bar{t}) = O(\sum_t m(t))$. Informally, given a memory capacity $m(t)$, we round $m(t)$ up to the nearest power-of-2 multiple of k_{min} (normalizing space) and then maintain memory constant at this normalized capacity for the minimum interval that guarantees time normalization – adding idle cycles if necessary, i.e. if $m(\cdot)$ would grow above the allotted capacity before the end of the interval.

More formally, denoting by $\bar{m}(\bar{t})$ the normalized memory profile function, we consider a set $\bar{t_0}, \ldots, \bar{t_n}$ of *upticks*, which are the only destination ticks in which $\bar{m}(\bar{t})$ *may* change (i.e. $\bar{m}(\bar{t_i}) = \cdots = \bar{m}(\bar{t_{i+1}} - 1)$), so $\bar{m}(\bar{t})$ is completely defined by its values on the upticks. We map the original clocktick 0 into $\bar{t_0} = 0$, and denote in general with t_i the original clocktick mapped into $\bar{t_i}$. Then, for any $i \geq 0$ we set $\bar{m}(\bar{t_i})$ to the lowest power-of-2 multiple of k_{min} no smaller than $m(t_i)$, i.e. $\bar{m}(\bar{t_i}) = k_{min} \cdot 2^{\lceil \log(\frac{m(t_i)}{k_{min}}) \rceil}$, and determine $\bar{t_{i+1}}$ as follows. Let t_i^+ be the earliest original clocktick, if any, after t_i such that $m(t_i^+) > \bar{m}(\bar{t_i})$. If there is no such t_i^+ equal to or smaller than $t_i + s \cdot \bar{m}(\bar{t_i})$, then either we are at the very end of the request sequence, or the original memory profile remains below $\bar{m}(\bar{t_i})$ for at least $s \cdot \bar{m}(\bar{t_i})$ ticks – in the latter case we simply map $t_i, \ldots, t_i + s \cdot \bar{m}(\bar{t_i})$ into $\bar{t_i}, \ldots, \bar{t_i} + s \cdot \bar{m}(\bar{t_i})$, and set $\bar{t_{i+1}} = \bar{t_i} + s \cdot \bar{m}(\bar{t_i})$ and thus $t_{i+1} = t_i + s \cdot \bar{m}(\bar{t_i})$). If we are at the very end of the request sequence, we simply imagine there is a sharp rise of memory consumption after it, and we set t_i^+ to be the first clocktick *after the completion* of the request

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

sequence with $m(t_i^+) = \infty$. Whether this is the case, or whether t_i^+ "occurred naturally", we map $t_i, \ldots, t_i^+ - 1$ into $\bar{t}_i, \ldots, \bar{t}_i + (t_i^+ - t_i) - 1$, as above, and map the single clocktick t_i^+ into the interval $\bar{t}_i + (t_i^+ - t_i), \ldots, \bar{t}_i + s \cdot \bar{m}(\bar{t}_i)$, with all clockticks in the interval save the last having memory $\bar{m}(\bar{t}_i)$ (thus achieving time-normalization). The last clocktick becomes the first clocktick t_{i+1}^- of the $(i+1)$-th interval, and is allotted memory equal to the lowest-power-2 multiple of k_{min} that is no smaller than $m(t_i^+)$.

It is easy to prove that space and time normalization does not increase the memory integral of the original profile by more than a constant factor. Due to space constraints, we defer this to the full version of the paper.

4.3 Compartmentalization An additional useful transformation we can apply to a (space- and time-normalized) memory profile, and implicitly to a generic page replacement algorithm, is *compartmentalization*. Compartmentalization expands the memory profile by adding to each box a prefix of duration equal to s times the capacity of that box, and a suffix of identical duration at the end of which all pages are evicted from memory. Note that compartmentalization can always be carried out online.

We can easily show that if an optimal page replacement algorithm services a request sequence over a sequence of boxes, it does the same even if the boxes are compartmentalized. Very simply, one can reload at the beginning of the box any pages in memory, or being loaded in memory, in the absence of compartmentalization; at s ticks per page, this takes no longer than the added prefix. Note that the original algorithm could have started the box with a pending fault, terminating after $s' \leq s$ ticks. If so, we simply idle for s' ticks, and then reproduce the original page replacement, ending with the memory exactly in the same state (including any pending fault). Then, we allow any pending fault to terminate (which takes at most s ticks), and idle through the rest of the suffix. The page replacement strategy above is not necessarily optimal, but it obviously provides an upper bound on the memory impact of the optimal strategy in the presence of compartmentalization. We then immediately have:

Proposition 1. *Under optimal replacement policy, compartmentalization increases the memory impact of a time-normalized memory profile by a factor of at most three.*

We stress that the above holds for the optimal replacement policy, not necessarily for *any* policy—in fact, it is not difficult to show that it does not hold for some online algorithms. But ultimately, we only care about bounds under optimal page replacement, because we know we can translate them into bounds under e.g. LRU with only $O(1)$-overhead (in terms of memory impact and resource augmentation).

The usefulness of compartmentalization lies in the fact that we can subsequently add between (expanded) boxes stretches of arbitrarily little memory without hindrance—we do so in Section 5.1. Note that such an addition is not necessarily an expansion, because the added intervals could sport lower capacity than both the boxes preceding them and those following them—so they could cause a loss of memory contents and a degradation of performance if they were added without compartmentalization. This is not the case with compartmentalization, since the memory contents are cleared at the end of each box anyways. Compartmentalization is a way to add these extra stretches knowing that they will at most triple the memory impact of a time-normalized memory profile. Compartmentalization also plays a crucial role in circumventing a subtle issue encountered in Section 6.2 that arises from the unexpected power of idling around the discontinuities of a memory profile function.

5 The Tight Relationship between Green Paging and Parallel Paging

Green paging and parallel paging appear to be two wildly different variants of paging, both in terms of system model (sequential processing with variable memory vs. parallel processing with constant memory) and in terms of goals (minimizing memory consumption over time vs. minimizing completion time).

In this section, we describe a black-box approach for turning online algorithms for green paging into online algorithms for parallel paging with the same competitive ratio. We then show a relationship in the other direction, which is that any lower bound construction for online green paging can be transformed in a black-box fashion into an online lower-bound construction for (deterministic) parallel paging.

For simplicity, we shall hereafter assume that p is a power of two. The same results can be obtained, with some additional complications, assuming that p is just an integer, substituting $\lceil \log p \rceil$ for $\log p$ in all formulas. All logarithms in this paper are to base 2.

5.1 Transforming Green Paging Algorithms into Parallel Paging Algorithms The key idea to translate an efficient algorithm for green paging algorithm into an efficient one for parallel paging is simply to generate independently an efficient green paging strategy for each of the p parallel sequences, and "pack" (expansions of) the memory profiles together into the shared memory, guaranteeing that (a) as little memory as possible is wasted in the process, including the expansion, and (b) at any given time, the integral of the memory in each profile so far is roughly the same. We remark that even though the use of normalization results in memory profiles that are sequences of "rectangular" boxes of space-time (to be packed within a large rectangle of space-time, with height equal to the amount of shared memory k, and width equal to the time taken to service the sequences), one cannot

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

exploit standard 2D packing results: one cannot treat each profile as a single enveloping rectangle without potentially wasting $\Theta(k)$ memory, but one cannot treat each profile as a *set* of rectangles to be packed independently either, since the individual boxes of each profile must obviously be placed after each other in a specific order and cannot overlap temporally.

We now describe this packing process. We assume the p green paging memory profiles we start with are already compartmentalized and space- and time-normalized. Effectively, given the partition of each memory profile into boxes, at any given time step we execute (one box of) one or more memory profiles in parallel, adding an idle cycle at capacity zero to all the other memory profiles. Note that different consecutive boxes of the same profile potentially take place in non-contiguous intervals of time.

At any given time, we assign highest priority to the profile without any allocated memory—i.e., the processor is idle or has just completed a box—that has incurred, so far, the lowest total memory impact (with ties broken arbitrarily); any memory page that becomes available is then *reserved* for that profile, until enough pages become available that its next box can be scheduled. No other profile with no allocated memory can be executed before, not even if enough room in memory is available. Clearly, priorities change dynamically, so if a processor q that just completed a box has currently the highest priority, then the space in memory is reserved for q. (Put another way: at any time step, if two processors p and q have no allocated memory and q has had a lower memory impact than p so far, then p does not get to run unless and until q gets (enough memory) to run.) Note that this process is online, since each box is scheduled without any knowledge of future boxes.

We now show that this process completes within $3c_{max}/k_{min}$ clockticks, where c_{max} is the maximum cost of any profile.

Lemma 1. *Consider p (space- and time-normalized, and compartmentalized) green paging strategies for p request sequences $\sigma_1, \ldots, \sigma_p$ with memory capacity between k and $k_{min} = k/p$, and let c_{max} be the maximum memory impact of any profile. Then, they can be packed online to yield a parallel paging strategy that completes within time $3c_{max}/k_{min}$.*

This immediately yields the following.

Theorem 8. *Consider p (space- and time-normalized, and compartmentalized) green paging strategies for p request sequences $\sigma_1, \ldots, \sigma_p$ with memory capacity between k and $k_{min} = k/p$, and assume that each is optimal with α resource augmentation within a factor of β, and that their respective costs are within a factor of γ of each other. Then they can be packed to yield a parallel paging strategy that completes, with α resource augmentation, within time $6\beta\gamma$ of any parallel paging strategy for the p sequences. Furthermore, if the green paging is online, so is the parallel paging.*

Proof. Let c_{min} be the minimum cost of any of the p green paging strategies, and $c_{max} \leq \gamma c_{min}$ be the maximum. Then, by Lemma 1, the p green paging strategies can be packed online into a single parallel paging strategy that completes within time $T = 3\gamma c_{min}/k_{min}$. Suppose there were a parallel paging strategy that completed all p sequences in time less than $T/6\beta\gamma$. This would automatically yield a green paging strategy for each of the p sequences, with memory capacity between k and 0 (note, not k_{min}). The least expensive of these strategies would then have cost less than $(k/p)\cdot(T/6\beta\gamma)$; raising the memory capacity to k_{min} whenever lower would increase the cost to less than

$$\frac{k}{p}\cdot\frac{T}{6\beta\gamma}+k_{min}\cdot\frac{T}{6\beta\gamma} = \frac{2k_{min}T}{6\beta\gamma} = \frac{2k_{min}}{6\beta\gamma}\cdot\frac{3\gamma c_{min}}{k_{min}} = \frac{c_{min}}{\beta},$$

against the hypothesis that the p strategies were all optimal within a factor of β for green paging with memory capacity between k and $k_{min} = k/p$. □

5.1.1 Analyzing Max, Average, and Median

Theorem 8 proves that, if one can come up with a green paging algorithm that is optimal within a factor of c with a certain resource augmentation, one can automatically obtain parallel paging *with the same resource augmentation* that is optimal within a factor of $O(c)$ *provided that the different sequences one is servicing incur green paging costs within a constant factor of each other*. In this case every sequence takes $\Theta(1)$ the time any other sequence takes to complete, and it is straightforward to prove that the median and average completion times are also within a factor of $O(c)$ of the optimal.

It is then also immediate to extend the analysis to a situation where one has p sequences of infinite length, and one is seeking a parallel paging algorithm ALG that minimizes, *simultaneously for every memory integral* w, the time necessary to complete all of the prefixes of the p sequences that could be completed, each in isolation, with a memory integral equal to at most w. The same strategy also obviously minimizes, within a constant factor, the average time for all these prefixes, and the median time.

The situation is different if one has *finite* sequences of drastically different cost. In this case, simply packing the corresponding green paging sequences can be a provably suboptimal choice for parallel paging, because one might be left after some time with only a few sequences uncompleted, whose green paging allocations are consuming only very little memory and "wasting" the rest—instead of using it in a way that *increases* the memory integral (potentially by $poly(p)$, as it would be easy to prove), but correspondingly

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

decreases the total completion time (again potentially by $poly(p)$, as it would be easy to prove).

In this case, we show that a simple variation on the scheme above that starts with a green paging algorithm with resource augmentation α and optimal within a factor of β, that is still online if the starting scheme is online, and provides a parallel paging strategy that, still with resource augmentation α, yields:

1. An average completion time that is optimal within a factor of $O(\beta)$.

2. For any $i \in \mathbb{N}$, a maximum completion time for all but a fraction 2^{-i} of all sequences that is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction 2^{-i-1}; this means:

   (a) a maximum completion time within a factor of $O(\beta \log p)$ of the optimal, and

   (b) a median completion time (intended as the time to complete at least $\frac{1}{2}p$ schedules) within a factor of $O(\beta)$ of the optimal time to complete at least $\frac{3}{4}p$ schedules.

It's not too difficult to show that, in terms of the inverse of the fraction of uncompleted sequences 2^i a) the logarithmic loss in time and b) the $O(1)$ "quantile augmentation" are both fundamentally inevitable in an online setting. So we can't do better. More precisely, quantile augmentation is necessary if you consider a set of sequences of exponentially growing length $2^p, (2^p)^2, \ldots, (2^p)^p$—basically because by the time you need only one more sequence completed to meet your quantile target, the work you've done so far is vanishingly small compared to what you still have to do, but because you do not know which sequence you have to focus on, you have to split your work evenly among all survivors, meaning that crucial sequence will receive only little space and take more time than necessary.

The scheme is simply to pack together the green paging strategies with memory between k and k/p for the p sequences, until $p/2$ of them have completed; then, pack together green paging strategies with memory between k and $\frac{k}{p/2}$ for the suffixes of the remaining $p/2$ sequences, until $p/4$ have completed; and so on, packing at each stage with $p/2^i$ "survivors" the green paging strategies with memory between k and $\frac{k}{p/2^i}$ for those survivors, until all sequences have completed. Recall that in each stage, we pack the next box of a processor that has the highest priority (whose memory impact is minimum so far). This is obviously an online strategy as long as the green paging strategies it is based on are online.

In the scheme above denote by T_i, with $i = 0, \ldots, \log(p)+1$, the first time when no more than $\frac{p}{2^i}$ sequences remain uncompleted, and by σ_{p+1-j}, with $1 \leq j \leq p$, the j-th sequence to complete – so that σ_1 completes *last*, σ_p completes *first*, and all of $\sigma_{2^i}, \ldots, \sigma_{2^{i-1}+1}$ complete between $T_{\lg(p)-i}$ and $T_{\lg(p)-i+1}$. The key idea is that $\sigma_{2^i}, \ldots, \sigma_{2^{i-1}+1}$ remain all uncompleted at time $T_{\lg(p)-i}$, but constitute at least one quarter of those sequences uncompleted at time $T_{\lg(p)-i-1}$, so that they "occupy" at least one quarter of the "memory space" in the interval $[T_{\lg(p)-i-1}, T_{\lg(p)-i}]$ – and if their memory occupation is optimal within a factor of β, writing for brevity $(T_j - T_{j-1})$ as ΔT_j, it would then be impossible to complete them all in time less than $\frac{1}{4\beta}\Delta T_{\lg(p)-i}$. In fact, σ_1 also occupies the *entire* memory space throughout the last interval $[T_{\lg(k)}, T_{\lg(p)+1}]$ (in addition to at least one quarter – in fact, at least one half – of the space in the interval $[T_{\lg(k)-1}, T_{\lg(p)}]$), and so it would be impossible to complete in time less than $\frac{1}{4\beta}\Delta T_{\lg(p)} + \frac{1}{\beta}\Delta T_{\lg(p)+1}$.

Then under *any* algorithm the average completion time of the p sequences is at least:

(5.1)
$$T_{avg} = \frac{1}{\beta}\Delta T_{\lg(p)+1} + \sum_{i=1}^{\lg(p)} \frac{1}{4\beta}\frac{p}{2^i}\Delta T_i > \sum_{i=0}^{\lg(p)} \frac{1}{4\beta}\frac{p}{2^{i+1}}\Delta T_{i+1}$$

while under the scheme above the average completion time is no more than:

$$\sum_{i=0}^{\lg(p)} \frac{p}{2^i}\Delta T_{i+1} < 8\beta T_{avg}. \tag{5.2}$$

Thus we have the following theorem:

Theorem 3. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging with competitive ratio $O(\beta)$ for average completion time. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

In a similar fashion, note that to complete all but a fraction $2^{-(i+1)}$ of *all* p sequences, any algorithm must complete for any $j \geq \lg p - i$ all but at most $2^{-(i+1)}p$ of $\sigma_1, \ldots, \sigma_{2^j}$, thus requiring time at least $\frac{1}{4\beta}\Delta T_{\lg p - j}$, since *each* of the 2^j sequences requires at least $\frac{1}{2^j\beta}\Delta T_{\lg p - j}$ space. Then, the total time $T_{2^{-(i+1)}}$ for any algorithm to complete all but a fraction $2^{-(i+1)}$ of all sequences satisfies

$$T_{2^{-(i+1)}} \geq \max_{j \leq i} \frac{1}{4\beta}\Delta T_{\lg p - j} \geq \frac{1}{4\beta i}\sum_{j=0}^{i} \Delta T_{\lg p - j}, \tag{5.3}$$

where the last term is no more than $\frac{1}{4\beta i}$ the time our scheme takes to complete all but a fraction 2^{-i} of all sequences.

Thus we arrive at the following theorem:

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Theorem 4. *Given an online algorithm for green paging with competitive ratio β, one can construct an online algorithm for parallel paging that achieves the following guarantee: For any $i \in \mathbb{N}$, the maximum completion time for all but a fraction 2^{-i} of all sequences is within a factor of $O(i\beta)$ of the optimal time to complete all but a fraction of 2^{-i-1}. Moreover, if the green paging algorithm uses α resource augmentation, then the parallel paging algorithm uses $O(\alpha)$ resource augmentation.*

5.2 Transforming Green Paging Lower Bounds into Parallel Paging Lower Bounds In this section, we consider the problem of transforming an arbitrary lower-bound construction for green paging into a matching lower-bound construction for parallel paging. Throughout the rest of the section, k denotes the maximum amount of memory that can be allocated in green paging, and $k/(2p)$ the minimum amount. We also use k to represent the amount of memory available in parallel paging, and p to be the number of processors.

Defining the notion of lower-bound construction for green paging We begin by formally defining the notion of a ***green paging lower-bound algorithm*** $\mathcal{L}$. A lower-bound algorithm $\mathcal{L}$ takes as input a deterministic online green-paging algorithm A (that uses $O(1)$ resource augmentation), and produces a request sequence $\sigma(A)$ on which A performs poorly. The way in which the lower-bound algorithm $\mathcal{L}$ and the green paging algorithm A interact is that the i-th request in $\sigma(A)$ is determined based on the behavior of algorithm A while serving the first $(i-1)$ requests in $\sigma(A)$.

Each lower-bound algorithm $\mathcal{L}$ must have a ***termination size*** R. This means that the request sequence $\sigma(A)$ terminates once the total memory impact incurred by A reaches R. Note that R is independent of the algorithm A.

A lower-bound algorithm $\mathcal{L}$ is said to ***achieve competitive ratio β*** if every green-paging algorithm A with $O(1)$ resource augmentation incurs a factor of $\Omega(\beta)$ more memory impact on $\sigma(A)$ than does the optimal green-paging algorithm[6]. That is, the optimal green paging algorithm incurs memory impact only $O(R/\beta)$.

We prove the following theorem.

Theorem 5. *Suppose there exists a green paging lower bound construction $\mathcal{L}$ that achieves competitive ratio $\Omega(\beta)$. Then all deterministic parallel paging algorithms (that use $\alpha \le O(1)$ resource augmentation) must incur competitive ratio $\Omega(\beta)$ for both average-completion time and makespan.*

Proof. Let A be a deterministic algorithm for parallel paging that uses resource augmentation $\alpha = O(1)$. We can assume without loss of generality that A always allocates space at least $k/(2p)$ to every processor. In particular, these minimum allocations combine to only use half of the memory, which up to a constant factor in resource augmentation can be ignored.[7]

As A executes the p processors on their request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$, each processor's request sequence σ_i is executed with some memory profile m_i. Since m_i always allocates between $k/(2p)$ and k memory, one can think of m_i as being a green-paging solution for sequence σ_i.

Given a lower-bound algorithm $\mathcal{L}$, we construct each of the request-sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$ by running parallel instances of $\mathcal{L}$, with resource augmentation 2α. The result is that each memory profile m_i incurs total memory impact R (where R is the termination size of $\mathcal{L}$ and is assumed without loss of generality to be sufficiently large). Moreover, if m_i^{OPT} is defined to be the memory profile that the optimal green-paging solution uses for request-sequence σ_i, then the total memory impact incurred by m_i^{OPT} is $O(R/\beta)$. Without loss of generality, each profile m_i^{OPT} is a box profile (i.e., it is normalized and compartmentalized). Because we are considering $\mathcal{L}$ with 2α resource augmentation (meaning that the algorithm against which $\mathcal{L}$ is competing has 2α resource augmentation), each profile m_i^{OPT} consists of boxes with heights between $\frac{k}{4\alpha p} = \Theta(k/p)$ and $\frac{k}{2\alpha} = \Theta(k)$.

We now consider the average completion time for Algorithm A. Since each processor has memory impact R, the first $p/2$ processors to complete must incur total memory impact at least $\Omega(pR)$, thereby incurring total running time at least $\Omega(pR/k)$. This means that the final $p/2$ processors to complete each take time more than $\Omega(pR/k)$. Thus $\Omega(pR/k)$ is a lower bound for both the average completion time and the makespan of A.

In order to complete the proof, we construct an alternative parallel-paging solution B that has makespan (and thus also average completion time) only $O(pR/k\beta)$. Because A has α resource augmentation, the amount of memory available to the parallel-paging algorithm B is only k/α.

We can assume without loss of generality that the profiles m_i^{OPT} are box profiles. The algorithm B performs the Box-Packing algorithm from Section 2.3 on the box profiles $m_1^{\text{OPT}}, \ldots, m_p^{\text{OPT}}$. In particular, whenever the total memory allocated to processors is less than $k/2\alpha$, algorithm B selects a processor i out of those not currently executing (if there is one) and allocates space for the next box the profile σ_i^{OPT}. Note that the box is guaranteed to fit into B's cache of size k/α, since the maximum box height in any profile σ_i^{OPT} is only $k/(2\alpha)$.

Whenever algorithm B is in a state where it has allocated at least $\frac{k}{2\alpha}$ memory to boxes, we call B ***saturated***, and

[6]Note that $\alpha = O(1)$ resource augmentation means that the optimal green-paging algorithm has minimum box-size $k/(2\alpha p)$ and maximum box size $k/(2\alpha)$.

[7]By allowing for an extra factor of two in resource augmentation, we can actually think of A as having $2k$ memory, k of which is pre-allocated evenly among the processors (note that giving A extra memory can only help it). By re-normalizing this new size to k, it follows that every processor has at least $k/(2p)$ memory at all times.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

whenever B has not allocated $\frac{k}{2\alpha}$ memory to boxes (because there are no more processors to allocate boxes to) we call B ***unsaturated***. The makespan (i.e., running time) of B can be broken into two components, the amount of time T_1 during which B is saturated, and the amount of time T_2 during which B is unsaturated.

Since the total memory impact of profiles $\sigma_1^{\text{OPT}}, \ldots, \sigma_p^{\text{OPT}}$ is $O(pR/\beta)$, the amount of time T_1 that B can spend saturated is at most $O(pR/\beta k)$ (recall that $\alpha \leq O(1)$ so α does not appear here).

On the other hand, whenever B is unsaturated, all of the remaining processors are executing simultaneously. It follows that T_2 is upper-bounded by the makespan of the processor $i \in \{1, 2, \ldots, p\}$ with the largest makespan (the makespan of a processor is the sum of the widths of the boxes in m_i^{OPT}). Each processor i incurs total memory impact $O(R/\beta)$ and has has minimum box-height $\Omega(k/p)$. It follows that the sum of the widths of the boxes in m_i^{OPT} is $O(R/\beta)/\Omega(k/p) \leq O(pR/\beta k)$.

Combining T_1 and T_2, the total makespan of algorithm B is at most $O(pR/\beta k)$. This is a factor of $\Omega(\beta)$ smaller than the average-completion time and the makespan for algorithm A, as desired. □

6 Tight Bounds for Green Paging

This section proves both lower and upper bounds for green paging. The equivalence results from the previous section immediately translate them into corresponding bounds for parallel paging.

6.1 Lower Bounds for Green Paging In this section we show lower bounds on the competitive ratio of deterministic algorithms. We start by showing a lower bound for the case $h = k$, that is, when we compete with an adversary with a fast memory as large as ours.

Theorem 9. *Let A be any deterministic online algorithm for the green paging problem where $h = k$ and s is the fetch time. Then, the competitive ratio of A is at least*

$$\min\left\{\frac{k-1}{4}, \frac{s}{2k}\right\}.$$

To force a high competitive ratio, in this case it is sufficient for the adversary to allocate all the k memory locations throughout the execution. Notice that our lower bound does not follow from the lower bound given for the special case of green paging studied in [18,21]—whereas the converse is true. Indeed, even if in both proofs the adversary runs the same algorithm, the fact that in their model faults are not "weighted"—that is, they cost the same irrespective of how many pages are in memory during the fault—prevents their argument to produce any non-trivial lower bound in the green paging model. For lack of space, the proof of this result is deferred to the full version of the paper.

The argument for the case $h = k$, however, fails when $h < k$. In that case, we will need a refined strategy to prove our lower bound; specifically, we will use a better approximation to OPT by letting the adversary choose more cleverly how many memory locations to allocate, rather than setting the capacity to h for the whole execution.

Theorem 2. *Suppose $s \geq p^{1/c}$ for some constant c. Consider the green paging problem with maximum box-height k and minimum box-height k/p. Let ALG be any deterministic online algorithm for green paging, and let α be the amount of resource augmentation. Then, the competitive ratio of ALG is $\Omega\left(\frac{\log p}{\alpha}\right)$.*

Proof. The proof proceeds as follows. First, we briefly recap the model parameters and some assumptions we make. Then, we consider for a generic online algorithm a specific request sequence, on which it is guaranteed to fault on every request, and that has some additional properties. Then, we show how an offline algorithm can service the same sequence paying, informally, α times the cost on a fraction at most $1/\log p$ of all requests – and $\alpha/\log p$ times the cost on all remaining requests.

Let us briefly recap the model parameters and assumptions. Let k be the maximum memory available to the online algorithm ALG, and k/p the minimum. ALG is compared to an offline algorithm OFF with memory between $h = k/\alpha$ and $k/(\alpha p)$, *that pays α times as much for the same space*: i.e., τ timesteps at memory size k/α cost OFF a total of $k\tau$, while they would cost ALG only $k\tau/\alpha$. Accessing a page costs ALG 1 timestep if the page is in memory. Otherwise, s timesteps are required to bring it into memory, and 1 more to access it.

We can assume without loss of generality that ALG is normalized and compartmentalized—since normalization and compartmentalization can be performed online, and increase the total cost by at most a constant factor. By the same token we assume for simplicity that k, p and α are powers of 2. We can also assume without loss of generality that ALG is "smooth-growing", in the sense that no box of capacity c is ever preceded by a box of capacity less than $c/2$; again, it is trivial to verify that any online algorithm can be transformed online into a smooth-growing one by increasing the total cost by at most a factor $O(1)$ (in fact, $4/3 = 1 + 1/4 + 1/16 + \ldots$).

Also, we can assume $s \geq p$: if s were smaller, we can simply modify an algorithm to never exceed a capacity threshold of more than s times the minimum – by simply substituting boxes of minimum capacity for any boxes exceeding the threshold. Then servicing the same requests in these boxes can take no more than s times longer, with a capacity that is at least s times smaller. Note that, this implicitly replaces p with s, meaning that the lower bound that we will get is $\Omega(\log s)$. Since $s \geq p^{1/c}$, this is $\Omega(\log p)$.

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Given ALG, we construct an evil request sequence σ as follows. First, we request $k+1$ distinct pages $p_0, p_1, \ldots, p_k$. Then, every further request is for the most recently evicted page; we extend the request sequence to include at least $(k+1)\log p/\alpha$ requests. It is immediate that ALG faults on every request (and the cost on the initial $k+1$ is a fraction at most $O(\alpha/\log p)$ of the total). Let $B(2^i)$ for $i = 1, 2, \ldots, \log k$, be the total cost budget spent by ALG on boxes of capacity 2^i (i.e., 2^i times their total duration). Obviously, the cost incurred by ALG c_{ALG} equals $\sum_i B(2^i)$.

Now, let $j = \arg\min_i B(2^i)$ for $(k/p)\log p < 2^i < k/\max(\alpha, \log p)$; note that $B(2^j) = O(c_{ALG}/\log p)$. We service σ with a normalized and compartmentalized algorithm alternating between capacity 2^j and the minimum capacity $k/(\alpha p)$. In particular, whenever ALG's capacity is at least 2^j, we adopt minimum capacity $k/(\alpha p)$; we refer to any maximal interval of such requests as an *island*. Whenever ALG's capacity is less than 2^j, we adopt the larger capacity 2^j; we refer to any such maximal interval of such requests as a *sea*. We refer to the first box of a sea, and of an island, as its *shore*.

We show that we can service any one sea with at most 2^j faults on its shore. We do so by loading and holding in our memory a copy of ALG's memory, and an *eviction stack* with the pages not in ALG's memory that have been most recently evicted by ALG (in particular, with the most recently evicted on top). It is immediate that after the initial setup phase (which takes place on the sea shore and incurs at most 2^j faults), every request will be for the page on the top of the stack—which is never empty since at sea ALG's capacity is strictly less than 2^j.

Note that the total number of sea and land shores differs by at most 1; and that, by the smooth-growth property, all island shores are boxes of capacity 2^j, preceded by a box of capacity 2^{j-1} Then, the total cost we incur on sea shores is at most $\alpha O(B(2^j))$, i.e. (by the definition of j) no larger than $O(\alpha/\log p)c_{ALG}$. Once offshore, we have capacity that is at most $O(p/\log p)$ that of ALG, and since we never fault and ALG always does, we incur a cost that is at most $O(\alpha/\log p)$ of ALG's. Similarly, on land requests, our capacity is $k/(\alpha p)$ while ALG's is at least $(k/p)\log p$. Thus, as we both fault on each land request, our cost is at most $O(\alpha/\log p)$ ALG's.

Then, we can service the entire request sequence with a cost that is $O(\alpha/\log p)c_{ALG}$. □

6.2 $O(\log p)$-Competitive Green Paging In this section we study (deterministic) algorithms for green paging. We start by showing that a natural algorithm is optimal to within a factor of two, for a large set of values of parameter s, when $h = k$, that is, when we compete with an adversary with a fast memory as large as ours.

However, such an algorithm seems too naive to perform well in practice, since it does not adapt enough to the locality experienced by the request sequence as it unfolds. This is confirmed by the analysis: the worst-case performance of this algorithm remains unchanged even if the adversary has a memory of size 1. Hence, later we design and analyze a more advanced online algorithm, with nearly optimal performance when provided with a factor of two of resource augmentation.

6.2.1 Warm-up: (k,k)-Memory Allocation The most natural idea for green memory allocation is to "switch off", by reducing the memory capacity, cache locations that contain pages whose next occurrence in the request sequence is very far in the future. This suggests the following online algorithm, which we call *Power-Down LRU* (PD-LRU).

Algorithm 1 PD-LRU

1. Acts like LRU, but always discards a page whose last occurrence in σ is at least s requests apart from the last serviced request.

The term Power-Down indicates that reducing capacity can be seen as powering down cache locations, thus provoking the loss of the pages that were contained in them. This simple algorithm dates back to the work of López-Ortiz and Salinger [21], where it is termed LRU_α, and which studies a special case of green paging. In fact, parameter α in their model is equivalent to parameter s in green paging; that is, PD-LRU $= \text{LRU}_s = \text{LRU}_\alpha$. The analysis, though, will require significantly more work. The proof that PD-LRU is $\min\{2k, s\}$-competitive is deferred to the full version.

6.2.2 (h,k)-Memory Allocation We now present a deterministic online algorithm, which we call BLIND, whose competitive ratio is $O(\log p)$ when provided with a factor of (at least) two of resource augmentation, that is, when $h \le k/2$. According to Theorem 2, this is optimal to within a constant factor when a constant amount of resource augmentation is given.

Algorithm BLIND is quite simple: it implements the LRU replacement policy, running with a suitably *predetermined* sequence of capacities. That is, the maximum amount of pages that BLIND retains in its cache at a given time step is *independent of* σ. Hence, not only BLIND does not use information about future requests in order to adjust its capacity (because it is an online algorithm): it doesn't even look at the *past* requests (whence the name *blind*)! The intuition is that, roughly speaking, BLIND, for each suitably defined period of time, "divides" its incurred cost among capacities $k/p, 2k/p, 4k/p, \ldots, k$ in such a way that at least a $\log p$-th fraction of its incurred cost is spent at the "right" capacity, that is, at roughly the same capacity that OPT has on

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited
Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

the same subsequence of requests.

The road to the specification and the analysis of BLIND is divided into three parts: first, we design an $O(1)$-approximation offline algorithm by building a "quantized" version of OPT; second, we obtain a simplified version of the above algorithm which achieves a logarithmic (in p) approximation; and lastly, we show that a non-clairvoyant version of the latter algorithm achieves the same approximation factor when provided with a cache of size at least twice. We begin with some necessary preliminaries.

Definition 6. *For integer i, an i*-phase *of an algorithm for the green paging problem is a sequence of $3s2^i$ consecutive time steps spent at capacity 2^i.*

Thus, the total memory impact incurred by an algorithm for an i-phase is $3s4^i$. We now define an ***$(i, k/p)$*-universal box profile**, a key concept in the design of our algorithm. Its definition is recursive.

Definition 7. *Let i be an integer and k/p be a power of two. A $(i, k/p)$*-universal box profile *of an algorithm for the green paging problem is a $\log k/p$-phase if $i \leq \log k/p$, and the concatenation of four consecutive $(i-1, k/p)$-universal box profiles followed by an i-phase otherwise.*

The memory impact (cost) of a $(i, k/p)$-universal box profile is easily established.

Lemma 2. *The memory impact of a $(i, k/p)$-universal box profile is $3s4^i(\log \left\lceil \frac{2^i}{k/p} \right\rceil + 1)$.*

We say that an algorithm A services a subsequence of consecutive requests σ_j *over an i-phase* when σ_j gets serviced in $3s2^i$ time steps with A using capacity 2^i, but where each time step t is charged for a cost of exactly 2^i, even if $P(t)$, the number of pages in cache at time t, is smaller. Broadly speaking, its cost is accounted as if its capacity were *exactly* 2^i for each time step. (This is reasonable in practice since a paging algorithm works with *boxes* of cache locations rather than single locations, and thus the "cost"—in terms of its energetic costs or in terms of space taken away from other processors—of a box should be accounted even when not all the locations of the box are used). Thus, a subsequence of consecutive requests serviced over an i-phase comes at a cost of exactly $3s4^i$. Finally, Let OPT_i, $i = 0, 1, \ldots, \log k$ be an optimal offline algorithm running with capacity at most 2^i. Obviously, $\text{OPT}_i(\sigma) \geq \text{OPT}_{i+1}(\sigma)$, since an offline algorithm does not need to use all its cache locations.

We now introduce BLOCK_i, a recursively-defined offline algorithm which well approximates OPT_i. BLOCK_i is defined as an offline algorithm that services an input sequence σ by servicing its longest possible prefix either over an i-phase (in this case we say that BLOCK_i *maximizes*) or by simulating BLOCK_{i-1} for the same cost $3s4^i$ as an i-phase (in this case we say that BLOCK_i *simulates*), whichever yields the *longest* prefix; after having serviced such a prefix, BLOCK_i flushes its memory and services the remaining suffix of σ in the same way. Note that BLOCK_i effectively partitions σ into subsequences $\sigma_1, \sigma_2, \ldots, \sigma_x$ each of them (with the possible exception of σ_x) serviced incurring cost $3s4^i$. We have the following result, which shows that BLOCK_i does not spend much more than OPT_i.

Proposition 2.

$$\text{BLOCK}_i(\sigma) \leq 64 \cdot \text{OPT}_i(\sigma).$$

Now we give an offline algorithm, IDLE-BLIND_i, which is a $O(\log p)$ approximation of BLOCK_i. This is an intermediate step towards both the definition and the analysis of our sought online algorithm, BLIND. IDLE-BLIND_i is an offline algorithm that services each such subsequence σ_j over an $(i+1, k/p)$-universal box profile (and thus with capacity 2^{i+1}), as follows. If BLOCK_i maximizes over σ_j, IDLE-BLIND_i idles (that is, it stops servicing requests) over the four initial $(i, k/p)$-universal box profiles of the $(i+1, k/p)$-universal box profile, services σ_j over the last $(i+1)$-phase applying a LRU replacement policy, and then possibly idles until the end of the $(i+1)$-phase. Otherwise, if BLOCK_i simulates over σ_j, IDLE-BLIND_i services σ_j with IDLE-BLIND_{i-1} over the four initial $(i, k/p)$-universal box profiles, and then possibly idles until the end of the $(i+1)$-phase. Notice that *the capacity of* IDLE-BLIND_i *is simply a sequence of $(i+1, k/p)$-universal box profiles, and thus is independent of the request sequence*, and in particular of the past requests. The next proposition bounds from above the performance of IDLE-BLIND_i.

Proposition 3.

$$\text{IDLE-BLIND}_i(\sigma) \leq 4(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{BLOCK}_i(\sigma).$$

Now observe that IDLE-BLIND_i *exploits its clairvoyance only to decide when to idle*. We define BLIND_i similarly to IDLE-BLIND_i, with the difference that BLIND_i never idles. Notice that BLIND_i is an online deterministic algorithm.

Intuitively, it appears that the use of idle cycles can only increase the cost incurred by a green paging algorithm, since to each idle cycle corresponds a cost equal to the capacity of the memory at that moment, but no progress on the service of the request sequence. However, this intuition is wrong—in sharp contrast with classic paging, where idling never helps. More details on this in the full version of the paper.

The unexpected power of idling around the discontinuities of a memory profile function is neutralized when memory profiles are compartmentalized. In fact, if capacity never changes there is clearly no advantage in idling; but when

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

it changes, roughly speaking, there is still no advantage in idling since the memory contents are cleared at the end of each box anyways. Hence, thanks to compartmentalization, and leveraging properties of the LRU replacement policy adopted by both BLIND_i and IDLE-BLIND_i, we have the following.

Proposition 4. *Assume compartmentalization. Then,*

$$\text{BLIND}_i(\sigma) \leq \text{IDLE-BLIND}_i(\sigma).$$

Define BLIND as $\text{BLIND}_{\log k/2}$. Below we provide its straightforward pseudocode.

Algorithm 2 BLIND

1. Service σ through a sequence of $(\log k, k/p)$-universal box profiles, evicting the least recently used page(s) whenever capacity is adjusted downwards or a page fault occurs.

Putting all pieces together, we obtain the following result.

Theorem 1. *Using resource augmentation $\alpha = 2$, the competitive ratio of BLIND is $O(\log p)$.*

Proof. Combining Proposition 1 with the results of [5],[8] Proposition 2, Proposition 3, and Proposition 4, we have

$$\begin{aligned}\text{BLIND}_i(\sigma) &\leq 3 \cdot \text{IDLE-BLIND}_i(\sigma) \\ &\leq 12(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{BLOCK}_i(\sigma) \\ &\leq 768(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1) \cdot \text{OPT}_i(\sigma).\end{aligned}$$

Hence, on a cache of size $k = 2^{i+1}$, and with resource augmentation $\alpha = 2$, the competitive ratio of BLIND_i is at most $768(\log \left\lceil \frac{2^{i+1}}{k/p} \right\rceil + 1)$, and since BLIND coincides with $\text{BLIND}_{\log k/2}$, the theorem follows. □

7 Conclusions and Open Problems

The fundamental result of this work is that green paging, with a memory capacity between k and k/p, is essentially equivalent to parallel paging, with memory capacity k and p processors; and obtaining almost tight bounds for both. It is interesting to note that, informally, an optimal parallel solution must also be "green"; this validates the folklore principle in "practical parallelism" that extra parallelism should never be purchased at a(n excessive) cost in work-efficiency.

A crucial difference from classic paging, or even pure page replacement with variable memory capacity, is that any online algorithm is at best $\Theta(\log p)$ competitive, even with $O(1)$ resource augmentation: i.e., informally, online memory allocation is much harder than online page replacement, at least in the presence of significant parallelism or significant capacity flexibility. Crucially, the source of this $\log p$ factor appears to be the lack of knowledge about future locality of the computation. In practice, future locality can be often gauged, whether by profiling or simply because it does not change too often during the course of a computation. It would be interesting to incorporate this knowledge into our model, showing if and to what extent it can rid us of the $\log p$ factor. Also, we strongly believe that the $\log p$ factor can in fact subsume the logarithmic factor in ε when translating an efficient green paging strategy into a parallel paging algorithm with a low completion time for the all but the εp slowest processors (and that eliminating one will eliminate the other); this is also an obvious avenue for future work.

We would also remark that to simplify our analysis we were not very parsimonious with constant terms. Reducing constants, both theoretically and experimentally, through better analysis and more sophisticated algorithms, would be crucial to the practicality of our (or in fact any) general scheme for both green and parallel paging.

Last but not least, our model for green paging is a special case of a more general *budget paging* one where "memory costs" and the goal is minimizing a computation's budget. The linear dependence (with upper and lower caps at k and k/p) that we consider is probably the simplest model, but not necessarily the best in all situations. For example, in many high-performance systems, a crucial constraint is that temperature should not climb above a certain threshold; this translates into a highly non-linear cost model. Leaving energy considerations aside, a virtualization provider might also offer extra memory at a premium, or at a discount. In this sense, we believe that this work is only the tip of a large iceberg in modelling non-classic paging in many cases of practical interest.

Acknowledgements

This work was supported in part by NSF grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, CCF-1439084, CCF-1733873, CCF-1527692, CCF-1725647, and XPS-1533644; by the US Air Force Research Laboratory under cooperative agreement number FA8750-19-2-1000; and by the University of Padova under project BIRD197859/19 and project "Internet of Things" (MIUR grant L.232 "Dipartimenti di Eccellenza"). W. Kuszmaul was also supported in part by an NSF Graduate Fellowship and a Hertz Foundation Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as repre-

[8] In the full version of the paper we provide a self-contained proof for the cost of compartmentalization with LRU. When LRU in applied, the prefix of duration equal to s times the capacity of the box need not be added, and as a result the overhead is only two rather than three.

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

senting 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] K. Agrawal, M. A. Bender, R. Das, W. Kuszmaul, E. Peserico, and M. Scquizzato. Brief announcement: Green paging and parallel paging. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 493–495, 2020.

[2] R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4):1290–1303, 2000.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] M. A. Bender, E. D. Demaine, R. Ebrahimi, J. T. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive analysis. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, 2016.

[5] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[7] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, pages 171–182, 1994.

[8] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412, 2007.

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

[10] R. Das, K. Agrawal, M. A. Bender, J. Berry, B. Moseley, and C. A. Phillips. How to manage high-bandwidth memory automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 187–199, 2020.

[11] E. Feuerstein and A. Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.

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

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), 2012.

[14] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. Elastic caching. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156, 2019.

[15] A. Hassidim. Cache replacement policies for multicore processors. In *Proceedings of 1st Symposium on Innovations in Computer Science (ICS)*, pages 501–509, 2010.

[16] S. Kamali and H. Xu. Beyond worst-case analysis of multicore caching strategies. In *Symposium on Algorithmic Principles of Computer Systems (APOCS21)*. SIAM, 2021.

[17] A. K. Katti and V. Ramachandran. Competitive cache replacement strategies for shared cache environments. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215–226, 2012.

[18] M. Khare and N. E. Young. Caching with rental cost and zapping. *CoRR*, abs/1208.2724, 2012.

[19] R. Kumar, M. Purohit, Z. Svitkina, and E. Vee. Interleaved caching with access graphs. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1846–1858, 2020.

[20] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[21] A. López-Ortiz and A. Salinger. Minimizing cache usage in paging. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*, pages 145–158, 2012.

[22] A. López-Ortiz and A. Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science conference (ITCS)*, pages 113–127, 2012.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[24] I. Menache and M. Singh. Online caching with convex costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 46–54, 2015.

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

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

[27] M. Scquizzato. *Paging on Complex Architectures*. PhD thesis, University of Padova, 2013.

[28] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[29] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41:1054–1068, 1992.

[30] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[31] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41:665–676, 1992.

[32] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174–183, 2009.

Downloaded 09/30/21 to 24.107.2.41 Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited