Paging and the Address-Translation Problem

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

Abhishek Bhattacharjee
Yale University
CT, USA
abhishek.bhattacharjee@yale.edu

Alex Conway
VMWare Research
CA, USA
aconway@vmware.com

Martín Farach-Colton
Rutgers University
NJ, USA
martin@farach-colton.com

Rob Johnson
VMWare Research
CA, USA
robj@vmware.com

Sudarsun Kannan
Rutgers University
NJ, USA
sudarsun.kannan@rutgers.edu

William Kuszmaul
MIT
MA, USA
kuszmaul@mit.edu

Nirjhar Mukherjee
UNC Chapel Hill
NC, USA
nirjhar@unc.edu

Don Porter
UNC Chapel Hill
NC, USA
porter@cs.unc.edu

Guido Tagliavini
Rutgers University
NJ, USA
guido.tag@rutgers.edu

Janet Vorobyeva
Stony Brook University
NY, USA
janet.vorobyeva@stonybrook.edu

Evan West
Stony Brook University
NY, USA
etwest@cs.stonybrook.edu

ABSTRACT

The classical paging problem, introduced by Sleator and Tarjan in 1985, formalizes the problem of caching pages in RAM in order to minimize IOs. Their online formulation ignores the cost of address translation: programs refer to data via virtual addresses, and these must be translated into physical locations in RAM. Although the cost of an individual address translation is much smaller than that of an IO, every memory access involves an address translation, whereas IOs can be infrequent. In practice, one can spend money to avoid paging by over-provisioning RAM; in contrast, address translation is effectively unavoidable. Thus address-translation costs can sometimes dominate paging costs, and systems must simultaneously optimize both.

To mitigate the cost of address translation, all modern CPUs have translation lookaside buffers (TLBs), which are hardware caches of common address translations. What makes TLBs interesting is that a single TLB entry can potentially encode the address translation for *many* addresses. This is typically achieved via the use of huge pages, which translate runs of contiguous virtual addresses to runs of contiguous physical addresses. Huge pages reduce TLB misses at the cost of increasing the IOs needed to maintain contiguity in RAM. This tradeoff between TLB misses and IOs suggests that the classical paging problem does not tell the full story.

This paper introduces the Address-Translation Problem, which formalizes the problem of maintaining a TLB, a page table, and RAM in order to minimize the total cost of both TLB misses and IOs. We present an algorithm that achieves the benefits of huge pages for TLB misses without the downsides of huge pages for IOs.

SPAA '21, July 6–8, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461814

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**; • **Theory of computation** → **Caching and paging algorithms**; **Bloom filters and hashing**.

KEYWORDS

virtual memory; address translation; TLB; paging; hashing; iceberg

ACM Reference Format:
Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. 2021. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3409964.3461814

1 INTRODUCTION

In the classical ***paging problem***, a sequence of page requests $p_1, p_2, \ldots$ must be serviced using a memory of size P pages [16, 18, 22, 47]. The cost of servicing a page request is 0 if the page is currently cached in memory. Otherwise there is a ***page fault*** and an ***IO*** must be performed, which means that the page must be fetched from disk to RAM (perhaps evicting another page) at a cost of 1.

Paging is critical to ***virtual memory*** systems, where programs reference pages by ***virtual page addresses***. When a page is cached in memory, it also has a ***physical page address*** in the range

$\{1, 2, \ldots, P\}$, specifying the location where it is actually stored. Every virtual address referenced by a program must be translated to a physical address by a process called ***address translation*** (***AT***). If a virtual page does not have a physical address, the page's data must be fetched from external storage, a physical page must be allocated (and potentially freed first), the physical page filled with the contents from storage, and assigned to that virtual address. Address translations are stored in an in-RAM dictionary called the ***page table***.

AT incurs such a significant cost on real computers that modern CPUs come with specialized hardware accelerators called ***translation lookaside buffers (TLBs)*** that cache part of the page table. ***TLB hits***, that is, successful lookups in the TLB, are fast, typically a single or small number of cycles [45]. In contrast, it can take hundreds or even thousands of CPU cycles to perform an address translation in the page table, when there is a ***TLB miss*** [8, 29].

Although the cost of AT is ignored in the paging problem, the cost can be high—and can even dominate paging costs—because *every memory reference* must undergo address translation, whereas page fetches may be rare. Moreover, one can avoid paging by purchasing more RAM, and this is generally considered money well spent. In contrast, TLBs have hit hard physical and power limits, making AT costs effectively unavoidable.

In this paper, we address the algorithmic problem of how to organize both the TLB and the physical-address assignment in order to simultaneously optimize the total cost of address translation and paging. We show that, by combining ideas from low-associativity paging, recent advances in hashing, and compression, one can achieve strong, provable guarantees on the costs incurred by both the TLB and the page fetches.

Trends in the cost of address translation. AT overheads are becoming more significant because of several hardware trends. First, TLBs are too small to cache the working sets of modern parallel programs. Second, the access patterns of emerging workloads, such as machine learning and graph analytics, are irregular and difficult to prefetch. The increasing prevalence of parallel programming has led to recent TLBs allowing multiple threads (and even applications) to have entries in the TLB simultaneously [28], meaning that the effective size of the TLB is smaller for each thread. Additionally, in cloud environments, which increase parallelism by using virtual machines, each memory reference undergoes two translations—once in the guest and once in the host—which actually *squares* the cost of a TLB miss in the worst case [7]. Whereas the aforementioned trends result in increased pressure on the TLB and higher TLB-miss costs, trends towards faster storage devices lower the cost of paging, which further increases the relative overhead of address translation.

Larger TLBs would have higher hit rates, but the size of TLBs is limited because it is expensive—in terms of time, transistors, and power [9]—to perform (parallel) hardware key-value lookups in tables with many entries. TLBs are so small that some workloads spend as much as 83% of their execution time on address translations [8] (see also [26, 27, 30, 48]).

The ubiquity of TLBs. TLBs, and hence TLB performance bottlenecks, are also becoming more ubiquitous because of hardware and software trends. Traditionally, peripherals such as GPUs and network cards accessed RAM via physical addresses, and hence had no TLBs. Newer devices are beginning to support virtual memory in order to support safe, concurrent access by mutually distrusting users, as may occur when two virtual machines are sharing a hardware peripheral. For instance, recent GPUs by both Nvidia and AMD include page tables and TLBs so that multiple, unrelated kernels can run concurrently on different compute elements in the GPU. Newer network cards support ***remote direct memory access (RDMA)***, in which the network card performs memory reads and writes based on incoming packets without going through the CPU [33, 51]. This is widely used to support concurrent access to memory in a cluster, and these cards have page tables and TLBs (albeit, with different names) in order to ensure the card performs only authorized memory accesses. And, of course, multi-core and multi-CPU systems can have per-core and per-CPU TLBs. The results in this paper apply to all these TLBs in a modern computer.

Huge pages and what makes TLBs interesting. What makes TLBs interesting is that, rather than caching data, they cache pointers to data. Notably, this means that a *single pointer* can potentially point to a *very large amount* of data.

Indeed, the main thrust of increasing the effectiveness of TLBs in systems design has been to use ***huge pages***, which are runs of pages that are contiguous in the virtual address space [26]. Critically, existing huge-page methods require the run of pages also be placed contiguously in RAM (i.e., physical memory), so that a single TLB entry can translate any address in any page that is included in the huge page.[1] In this case, the TLB is used as a key-value store in which the keys are virtual addresses of huge pages (rather than of standard-size pages) and the values are physical addresses of huge pages (rather than of standard-size pages).

We call the set of page translations that a TLB entry encodes its ***coverage***. If the coverage of a TLB entry forms a contiguous run of virtual addresses defined by the high-order bits of the virtual addresses so encoded, we say that that entry encodes a ***virtual huge page***. If, additionally, the corresponding physical pages are stored contiguously in *physical memory*, then we say those pages form a ***physical huge page***.

Virtual and physical huge pages, the good and the bad. Virtual huge pages are an effective technique for reducing TLB misses [30, 35], not merely because they increase the coverage of each individual TLB entry, but also because they translate a contiguous run of virtual addresses. Programs that exhibit spacial locality in their memory-access patterns benefit from the large coverage of each huge page.

On the other hand, physical huge pages *increase* IO costs for three reasons:

[1] In our discussion, we elide many details of huge pages and TLBs, such as that most systems that implement huge pages use different TLBs for each size [15, 54] and only between one and three sizes are allowed, depending on the implementation. The algorithmic problems are the same, whether we are considering TLBs in the wild or the semi-domesticated TLBs described here.

(1) *Page-fault amplification.* In order to represent a collection of pages as a physical huge page, whenever any page within a huge page is fetched from disk, all the constituent pages must also be fetched. This turns what would be an IO for a single block into IOs for many blocks.
(2) *Reduced RAM utilization.* A physical huge page stores all the pages in its range, even if some are not accessed. This wastes RAM on pages that are not frequently referenced, thus leading to more page faults on pages that are more frequently referenced.
(3) *Fragmentation.* To mitigate these drawbacks, systems generally use a mix of regular and huge pages. Pages in a huge page are stored contiguously in RAM. To make room for them, any (non-huge) pages in the way must be evicted to disk, which can lead to IOs later when those evicted pages are re-accessed. [2]

In summary, huge pages come with a tradeoff: *virtual* huge pages enable a reduction in TLB misses, but *physical* huge pages cause an increase in IOs. There is a vast architecture and operating systems literature on optimizing the benefits and costs of huge pages [21, 30, 32, 35–37, 42]; for experiments that illustrate this tradeoff, see Section 6.

The full cost of address translation. In order to fully quantify the cost of address translation, one must consider TLB misses and IOs together. We call the software/hardware algorithm that manages the TLB, the page table, and the layout of pages in RAM a ***memory-management algorithm***.

To measure the cost of a memory-management algorithm we introduce the ***address-translation cost model***: each IO costs 1, each TLB miss costs $\varepsilon \in (0, 1)$, and each TLB hit costs 0.

Huge-page decoupling: all the virtual with none of the physical. In this paper, we are interested in designing memory-management algorithms that enjoy the TLB advantages of virtual huge pages without the IO costs of physical huge pages. A natural approach, which we call ***huge-page decoupling***, is to break up the value part of every TLB entry into an array of physical addresses: the ith entry of the array encodes both whether the ith page in the huge page is in RAM, and if so, where the ith page resides in RAM. Huge-page decoupling would mitigate or eliminate the disadvantages of physical huge pages: it would increase RAM utilization by only storing pages that the paging algorithm deems useful; it would reduce page-fault amplification by reducing the footprint of each huge page; and it would eliminate fragmentation by obviating the requirement that the constituent physical pages of a virtual huge page be contiguous.

A priori, huge-page decoupling is not viable because the TLB value does not have enough bits to store such an array. Indeed, current TLBs are designed to store one physical address of $\log P$ bits. In general, we shall use w to denote the number of bits used for each TLB value, and we shall treat w as being set by hardware.

[2] Rather than evicting pages to disk, one could also try to defragment those pages in RAM [30, 32]. The challenge in practice is that the performance overheads of defragmentation, even in memory, can easily exceed the performance benefits of huge pages.

Better encodings through low associativity. We show that huge-page decoupling schemes actually *are* possible by compressing the array of physical addresses as follows. Call a paging algorithm L-***associative*** if each page has L possible locations where it can be stored. If L is small, and we use only $O(\log L)$ bits per physical page address, we can store multiple physical page addresses per TLB value.

Intuitively, low associativity may result in all L locations for a page being occupied by pages that the paging algorithm would prefer to keep. If this happens, then the paging algorithm must evict one of the pages, resulting in extra (and otherwise unecessary) IOs. Hence, this low-associativity approach *also* appears at first sight to be a dead end: we replace the IOs needed for physical huge pages with the IOs needed for low associativity.

This paper. We show how to transform any paging algorithm into a low-associativity paging algorithm without increasing the IOs, while using minimal resource augmentation. Using this transformation, we can implement huge-page decoupling in order to realize the benefits of virtual huge pages without the need for physical huge pages.

Our main theorem is that we can simultaneously match the TLB miss rate of any memory-management algorithm (even one that only cares about minimizing TLB misses) while matching the IOs of any other memory-management algorithm (even one that only cares about minimizing IOs).

Section 2 discusses the results in more depth. The results lay down the theoretical groundwork for huge-page decoupling. In concurrent work, we are prototyping our techniques in cycle-accurate simulators and actual TLBs.

2 RESULTS AND TECHNICAL OVERVIEW

This section gives a detailed overview of the results and main technical ideas in the paper.

Section 3: Huge-page decoupling. Section 3 formalizes ***huge-page decoupling***. Recall that a huge-page decoupling scheme encodes in the TLB value for a huge page, all of the information of *which* of its constituent physical pages are present in RAM, and *where* those pages are located. The guarantee of a huge-page decoupling scheme is that the TLB can use virtual huge pages of some large ***huge-page size*** $h_{\max}$ pages, and RAM can be allocated at the granularity of normal-sized pages. That is, the choice of which normal-size pages are in memory can be made independently of the choice of which virtual huge pages are in the TLB.

Our approach to implementing huge-page decoupling is to treat RAM as a low-associative cache; we will avoid increasing IO cost by making use of a small amount of resource augmentation. That is, we equip a huge-page decoupling scheme with a ***resource-augmentation parameter*** δ, and the huge-page decoupling scheme may assume that there are never more than $(1 - \delta)P$ pages stored in RAM at a time.

The first goal of a huge-page decoupling scheme is to achieve a value of $h_{\max}$ that is as close as possible to the number w of bits that are used for each TLB value. Naturally $h_{\max}$ cannot be arbitrarily

large. There is a natural upper bound of

$$h_{\max} \leq w, \tag{1}$$

since each TLB value must use $h_{\max}$ bits to encode what subset of the pages in a huge page are present in memory.

The second goal of a huge-page decoupling scheme is to minimize the resource-augmentation parameter δ. If δ is small and the huge-page size $h_{\max}$ is large, then huge-page decoupling allows us to have large virtual huge pages at essentially no cost.

In this paper, we are able to come remarkably close to meeting the upper bound (1) on $h_{\max}$, achieving

$$\begin{cases} h_{\max} = \Theta(w/\log\log\log P) \\ \delta = o(1). \end{cases} \tag{2}$$

Recall that P is the number of pages that fit in physical memory; another interpretation of (2) is that, for each of the $h_{\max}$ physical pages that a TLB entry points to, we can encode the physical location of that page using only $\Theta(\log\log\log P)$ bits.

Section 4: Low-associativity paging and compact TLB encodings. In Section 4, we address the main challenge in designing a huge-page decoupling scheme, which is how to encode all of the information that we wish to store in each TLB value using only w bits. Our huge-page decoupling scheme must be able to use just $w/h_{\max}$ bits to encode the location of each page in a huge page.

A central technical idea in our TLB encodings is to re-purpose a classic technique in caching: low associativity. The idea of low associativity is to break the cache into small bins, hash each cache entry to a random bin, and then manage each bin individually via a paging algorithm, such as LRU. The bin size is referred to as the ***associativity*** (or ***set-associativity***) of the cache. Typically, the purpose of low-associativity caching is to simplify the task of implementing a cache (especially in hardware caches).

In this paper, we use low-associativity caching in a starkly different way. By limiting the number of options for where each page can be placed in physical memory, the physical page addresses in the TLB can also be encoded using very few bits. At the same time, the associativity must be large enough that, whenever a page is brought into memory, there is a legal position where it can be placed (i.e., a free position in the right bin). As a warmup result, we show that by setting the associativity to be $\tilde{\Theta}(\log P)$, and using resource augmentation $\delta = o(1)$, we can construct a simple huge-page decoupling scheme with virtual huge pages of size $h_{\max} = \Theta(w/\log\log P)$.

A key insight of this paper is that we can use recent advances in the design and analysis of balls-and-bins games to achieve an even smaller associativity. By employing the Iceberg$[d]$ balls-in-bins strategy, we show how to construct a huge-page decoupling scheme with $h_{\max} = \Theta(w/\log\log\log P)$ and $\delta = o(1)$.

Section 5: Optimizing the cost of address translation. Finally, we consider the task of optimizing the total TLB and IO cost of a memory-management algorithm on a sequence of page requests. For any memory-management algorithm $\mathcal{Z}$ and sequence of page requests $\sigma = (p_1, p_2, \ldots, p_n)$, let $C(\mathcal{Z}, \sigma)$ denote the total cost of $\mathcal{Z}$ in the address-translation cost model, let $C_{\text{TLB}}(\mathcal{Z}, \sigma)$ denote the total cost incurred due to TLB misses, and let $C_{\text{IO}}(\mathcal{Z}, \sigma)$ denote the total cost incurred due to IOs. In order to minimize $C(\mathcal{Z}, \sigma)$ we must simultaneously optimize $C_{\text{TLB}}(\mathcal{Z}, \sigma)$ and $C_{\text{IO}}(\mathcal{Z}, \sigma)$.

We prove that the problem of optimizing $C_{\text{TLB}}(\mathcal{Z}, \sigma)$ can be separated from the problem of optimizing $C_{\text{IO}}(\mathcal{Z}, \sigma)$, in the following sense. Let $\mathcal{X}$ and $\mathcal{Y}$ be arbitrary memory-management algorithms, each of which is allowed to use any mixture of huge-page sizes between 1 and $h_{\max}$. The only constraint on $\mathcal{X}$ and $\mathcal{Y}$ is that they operate on a physical memory of size $(1-\delta)P$ (rather than the full physical memory of size P). Using huge-page decoupling, we construct a new memory-management algorithm $\mathcal{Z}$ with the following guarantee. With high probability in P, the total cost of $\mathcal{Z}$ satisfies

$$C(\mathcal{Z}, \sigma) \leq C_{\text{TLB}}(\mathcal{X}, \sigma) + C_{\text{IO}}(\mathcal{Y}, \sigma) + \frac{n}{\text{poly}(P)}. \tag{3}$$

Importantly, even if $\mathcal{X}$ minimizes TLB misses (by using huge pages) and $\mathcal{Y}$ minimizes IOs (by not using huge pages), then $\mathcal{Z}$ combines the best performance features of $\mathcal{X}$ and $\mathcal{Y}$. The additive term in (3) says that a vanishingly small fraction $1/\text{poly}(P)$ of page accesses are permitted to be page faults in $\mathcal{Z}$ despite not being page faults in $\mathcal{Y}$.

Experiments and related work. Section 6 illustrates experimentally the IO-versus-TLB-miss tradeoff between virtual huge pages and physical huge pages. Section 7 discusses related work in depth.

3 HUGE-PAGE DECOUPLING

The idea behind huge-page decoupling is to enable the use of huge pages in the TLB, while letting the paging algorithm operate on normal pages. This will work by encoding the physical page addresses in the TLB entry for a given virtual huge page.

More precisely, a huge-page decoupling scheme takes as input a page replacement policy for RAM (the ***RAM-replacement policy***) and a huge-page replacement policy for the TLB (the ***TLB-replacement policy***). It consists of a RAM-allocation scheme that reduces the associativity of page placements in RAM, and an encoding/decoding scheme for translating between TLB values and physical addresses. All of these components must interact carefully. For example, the RAM-allocation scheme not only must achieve low associativity but also must be amenable to fast encoding and decoding; and unlike a standard TLB, which only covers virtual addresses that are mapped in physical memory, the encoding/decoding scheme must specify if a page is mapped or not. With so many moving parts, we take this section to carefully define all components of the system and their requirements before moving on to our main theorems.

Recall that the goals of a huge-page decoupling scheme are to maximize the size $h_{\max}$ of huge pages in the TLB, and minimize the resource-augmentation parameter $\delta \in (0, 1)$ for RAM.

The input replacement policies. Let V be the number of pages in virtual memory and P be the number of pages in physical memory. A ***virtual page address*** is any element of $[V] = \{1, 2, \ldots, V\}$. A ***physical page address*** is any element of $[P] = \{1, 2, \ldots, P\}$. A ***virtual huge-page address*** is any element of $[V/h_{\max}]$ (we assume $h_{\max}$ divides V). We use ℓ to denote the number of entries in the TLB, and w to denote the number of bits in each TLB value.

The RAM-replacement policy determines which virtual page addresses are in RAM at any given moment; we refer to the set of such addresses as the ***active set*** $\mathcal{A} \subseteq [V]$. The only restrictions on the RAM-replacement policy are that $|\mathcal{A}| \leq (1-\delta)P$ at all times, and that it is oblivious to the state and operation of the huge-page decoupling scheme.

The TLB-replacement policy determines which virtual huge-page addresses are in the TLB at any given moment; we refer to the set of such addresses as $\mathcal{T} \subseteq [V/h_{\max}]$. The only restrictions on the TLB-replacement policy are that $|\mathcal{T}| \leq \ell$ at all times, and that it is oblivious to the state and operation of the huge-page decoupling scheme.

The huge-page decoupling scheme. A huge-page decoupling scheme is an algorithm with three parts: a ***RAM-allocation scheme***, a ***TLB-encoding scheme***, and ***TLB-decoding scheme***.

The RAM-allocation scheme determines the physical address for each page fetched by the RAM-replacement policy. At any given moment in time, we use $\phi : \mathcal{A} \to [P]$ to denote the physical address $\phi(v)$ corresponding to each virtual page address $v \in \mathcal{A}$. The RAM-allocation scheme gets to decide the value of $\phi(v)$ whenever a new page is added to $\mathcal{A}$ by the RAM-replacement policy. The only restrictions on the RAM-allocation scheme are that ϕ must always be an injection and that ϕ must be ***stable***—that is, once a virtual page $v \in \mathcal{A}$ is assigned a physical address $\phi(v)$, that address cannot change until v is removed from $\mathcal{A}$.

The TLB-encoding scheme determines the w-bit TLB value for each virtual huge page in the TLB. At any given moment in time, we use $\psi : \mathcal{T} \to [2^w]$ to denote the current set of TLB values. The value of $\psi(v)$ is set (resp. unset) by the TLB-encoding scheme when the TLB-replacement policy inserts (resp. removes) v from $\mathcal{T}$. And the value of $\psi(v)$ is updated by the TLB-encoding scheme whenever any of the constituent virtual page addresses u of v are added or removed from $\mathcal{A}$ by the RAM-replacement policy.

The TLB-decoding scheme translates TLB values into physical addresses via a ***TLB-decoding function*** $f : ([V] \times [2^w]) \to ([P] \cup \{-1\})$. The TLB-decoding function must offer the following guarantee: If u is a virtual huge-page address in $\mathcal{T}$, and v is a virtual page address contained in u, then

$$f(v, \psi(u)) = \begin{cases} \phi(v) & \text{if } v \in \mathcal{A} \\ -1 & \text{otherwise.} \end{cases} \tag{4}$$

In other words, for every virtual page address v that is both contained in u and is in the active page set $\mathcal{A}$, the TLB-decoding function must be able to recover the physical page address $\phi(v)$ associated with v. And for every virtual page address v that is contained in u but is not in the active page set $\mathcal{A}$, the decoding function must encode the fact that v is not in $\mathcal{A}$ by returning the null address -1.

The TLB-decoding function f is determined once at the beginning of time and cannot be subsequently changed. The function f is permitted to be randomized (and thus can read the random bits used by our algorithm).

Constant-time high-probability decoupling schemes. A huge-page decoupling scheme is said to be ***constant time*** if, each time that the TLB-replacement policy modifies $\mathcal{T}$ or the RAM-replacement policy modifies $\mathcal{A}$, the huge-page decoupling scheme spends time $O(1)$ updating ϕ and ψ; and if the TLB-decoding function f can be evaluated in time $O(1)$.

In order to establish probability bounds over arbitrarily long sequences of requests for RAM-allocation schemes that have less than full associativity, we must deal with what happens when page v experiences a ***paging failure***, that is, when it is added to $\mathcal{A}$ by the RAM-replacement policy but cannot be assigned a physical address by the RAM-allocation scheme. The paging failure associated with v lasts until the RAM-replacement policy evicts v. We use $\mathcal{F} \subseteq \mathcal{A}$ to denote the set of virtual page addresses on which paging failures are occurring at a given moment. A randomized huge-page decoupling scheme is said to succeed ***with high probability in*** P if, at every point in time, the probability that $|\mathcal{F}| > 0$ is at most $1/\text{poly}(P)$. Later in the paper, when we use huge-page decoupling to construct efficient memory-management algorithms, we will handle paging failures by temporarily bringing the affected page into RAM whenever it is needed, and then allowing the page to subsequently be paged back out to disk.

4 LOW-ASSOCIATIVITY PAGING AND COMPACT TLB ENCODINGS

The key challenge in designing a huge-page decoupling scheme is to limit the associativity of the RAM-allocation scheme, so that each page in RAM has only a small number of options for where it can reside. At the same time, we must support an arbitrary RAM-replacement policy (i.e., the paging algorithm for managing which pages are in RAM), whose only constraint is that it never places more than $(1-\delta)P$ pages in RAM at a time.

In order to limit the associativity of the RAM-allocation scheme, we partition RAM into n buckets, each one comprising $B = P/n$ consecutive pages. To place a page in RAM, we randomly choose k buckets by computing k hash functions of the virtual page address; we select one of the buckets; and we place the page in some free slot within the chosen bucket. This yields an associativity of kB.

The bucket size B controls a trade-off between associativity and IO complexity: the smaller the B, the more likely it is for a page to find all of its k chosen buckets already full.

We show that, surprisingly, any (oblivious) RAM-replacement policy can be implemented with a low-associativity RAM-allocation scheme, using a small amount of resource augmentation. Our main theorem in this section is that this can be attained using $k = 3$ hash functions and buckets of size $B = \tilde{\Theta}(\log\log P)$, ultimately leading to a decoupling scheme that achieves $h_{\max} = \Theta(w/\log\log\log P)$ and $\delta = o(1)$.

We begin the section by showing, as a warmup, how to achieve $h_{\max} = \Theta(w/\log\log P)$ using $k = 1$ hash functions. We then extend the result to use $k = 3$ in order to achieve the stronger bound of $h_{\max} = \Theta(w/\log\log\log P)$.

Balls-and-bins games. We model RAM-allocation algorithms as ***dynamic balls-and-bins games***. In our balls-and-bins game, there are n bins, and there is an adversary that specifies an arbitrary sequence of ball insertions and deletions (and perhaps re-insertions), such that there are never more than m balls in the system. On each

insertion, a ball is thrown into some bin according to a rule that randomly chooses k bins and places the ball in one of them. The goal is to design the placement rule, such that it minimizes the maximum load across all bins. Importantly, the adversary is oblivious to the game's randomness; otherwise it could force all balls to go to the same bin.

The relationship between RAM-allocation schemes and balls-and-bins games is as follows. Each bin represents a bucket in RAM, and each ball represents a page. The adversary is the RAM-replacement policy (and the sequence of page requests), and the balls insertions/deletions correspond to page insertions/deletions in $\mathcal{A}$. Based on this analogy, we can use n and m in both contexts. We will use $\lambda = m/n$ to denote the (maximum allowable) average occupancy of the bins.

Observe that not every balls-and-bins game models a RAM-allocation scheme—it has to be ***online*** (i.e., balls are sequentially placed before seeing future requests) and ***stable*** (i.e., balls are not moved around once inserted). Both of these features are required in a huge-page decoupling scheme: page requests (and, thus, TLB and paging operations) are served in an online fashion, and the physical address of a page must not be changed until the page is swapped out.

The difficulty of reducing associativity. Suppose that a single hash function is used (i.e., $k = 1$) and that buckets have size $B = 1$, so that the associativity is 1. Then, physical addresses do not require any bits at all—virtual addresses are translated simply by computing their hash value, and thus no translations need to be cached in the TLB. The problem, of course, is that this configuration lends to a prohibitively large number of paging failures (recall that huge-page decoupling schemes must incur *no* paging failures with high probability in P, at any given point in time). To quantify this statement, consider a sequence of P distinct page accesses, starting from an empty RAM. By a standard balls-and-bins argument, where balls represent pages and bins represent page slots in physical memory (the unit-sized buckets), approximately P/e slots remain unused, with high probability in P. Thus, any paging algorithm that doesn't evict pages during the first $(1-\delta)P$ insertions (e.g., LRU, FIFO, etc.) will incur at least $(1/e-\delta)P$ paging failures with high probability in P. For $\delta = o(1)$, this is $\Omega(P)$ paging failures with high probability in P.

Achieving associativity $\tilde{\Theta}(\log P)$ with $k = 1$. Let $m = (1-\delta)P$ be the maximum number of pages that the RAM-replacement policy can cache simultaneously. We specify the bin size B and δ (and, thus, also m and n) below. For now, we use $k = 1$, which means that each ball is simply assigned to a random bin. In order so that no bins overflow, we must set the bin size B to be large enough that the maximum load of any bin is at most a $1+\delta$ factor larger than the average load. On the other hand, subject to bins not overflowing, we want B as small as possible to obtain a small associativity.

Since $k = 1$, at any given moment, the maximum load is

$$\begin{cases} (1+o(1))\frac{\log n}{\log(\log n/\lambda)} & \text{if } 1 \le \lambda = o(\log n) \\ \Theta(\lambda) & \text{if } \lambda = \Theta(\log n) \\ \lambda + O(\sqrt{\lambda \log n}) & \text{if } \lambda = \omega(\log n), \end{cases} \tag{5}$$

with high probability in n [44]. Thus, bin sizes B that allow for a $\delta = o(1)$ are in the third case.

Set the number of bins to be $n = m/(\log P \log\log P)$, so that the average load is $\lambda = \log P \log\log P = \omega(\log n)$. Then, with high probability in n (and thus P), the maximum load is

$$\begin{aligned} \lambda + O(\sqrt{\lambda \log n}) &= \lambda + O(\log P \sqrt{\log\log P}) \\ &= \lambda(1+\delta), \end{aligned}$$

where $\delta = O(1/\sqrt{\log\log P})$.

Note that the bucket size satisfies

$$B = \frac{P}{n} = \frac{P}{m}\cdot\lambda = \frac{P}{(1-\delta)P}\cdot\lambda = \frac{1}{1-\delta}\cdot\lambda > (1+\delta)\lambda,$$

which means that B is at least as large as the maximum load of the balls-and-bins game. Thus, every page fits in RAM at any fixed point in time, with high probability in P. Since addresses have size $\log B = \Theta(\log\log P)$, we get a decoupling scheme with huge-page size $h_{\max} = \Theta(w/\log\log P)$.

Theorem 1. *There exists a constant-time huge-page decoupling scheme using resource augmentation $\delta = o(1)$ that supports huge-page size $h_{max} = \Theta(w/\log\log P)$ with high probability in P.*

Proof. Recall that a huge-page decoupling scheme consists of three parts: a RAM-allocation scheme, a TLB-encoding scheme, and a TLB-decoding scheme. By having the RAM-allocation scheme use the balls-and-bins strategy described above, we ensure that each page has at most B positions where it can reside, where $B = \Theta(\log P \log\log P)$.

The TLB-encoding and decoding schemes can treat each TLB value as an array of $\Theta(\log\log P)$-bit elements $a_1, a_2, \ldots, a_{h_{\max}}$. If v is the ith page in the huge page represented by the TLB entry, and v hashes to bin j, then a_i indicates the position in bin j where v resides (or -1 if v is not in $\mathcal{A}$). Note that the huge-page decoupling scheme is easily made constant time by maintaining a hash table that keeps track of what the current value of $\psi(u)$ should be for each virtual huge page u that has at least one constituent page in RAM.

Our final task is to analyze the size of the failure set $\mathcal{F}$. For the sake of analysis, whenever a ball is inserted into a bin, label the ball as ***failed*** if the ball is inserted into a bin that already has B other balls (that are not labeled as failed). Note that the ball retains its failed label even if subsequently the load of the bin falls below B. From the perspective of the balls-and-bins game, failed balls are like any other balls. On the other hand, for the huge-page decoupling scheme, failed balls correspond to paging failures. That is, $|\mathcal{F}|$ is equal to the number of balls in the system that have the failed label. At any given moment, there are up to $m = O(P)$ balls $b_1, \ldots, b_m$ present. For each i, when b_i was inserted it had a $1/\text{poly}(P)$ probability of being labeled as failed. By a union bound, it follows that $\Pr[|\mathcal{F}| > 0] \le m/\text{poly}(P) = 1/\text{poly}(P)$, as desired. □

Achieving associativity $\tilde{\Theta}(\log\log P)$ with $k = 3$. A natural way to try to improve the associativity further is to use the balls-and-bins rule known as Greedy[2], in which each ball chooses 2 bins independently at random, and the ball is placed in the less full bin.

With this rule, the maximum load at any moment is at most

$$O(\lambda) + \log\log n + O(1), \tag{6}$$

with high probability in n [49]. This approach fails because the difference between the average load λ and the bound on the maximum load is $\Omega(\lambda)$, no matter what we choose λ to be.[3] Therefore, this forces the use of $\delta = \Omega(1)$ resource augmentation. Using Greedy$[d]$ for $d > 2$ doesn't help the situation, because the maximum load still grows as $O(\lambda)$ rather than λ.

Until recently, no balls-and-bins strategy was known to be simultaneously online, stable, and to have a maximum load of $(1+o(1))\lambda + O(\log\log n)$. The authors of this paper have another paper under submission that presents a balls-and-bin rule that has all of these features [34]. The rule, which is called Iceberg$[d]$, chooses $d+1$ bins per ball. In the case of $d = 2$, it attains the following bound.

Theorem 2 ([34]). *With high probability in n, at any fixed point in time the maximum load of Iceberg$[2]$ is at most*

$$(1 + o(1))\lambda + \log\log n + O(1),$$

in the dynamic setting, against any oblivious adversary.

For concreteness, we sketch out Iceberg$[2]$ here. Balls are placed into bins using three independent hash functions h_1, h_2, h_3. When inserting a ball x, we first look at bin $h_1(x)$ and insert the ball there if it is not too full. Otherwise, we use h_2 and h_3 to insert via Greedy$[2]$.[4] Intuitively, the reason that Iceberg$[2]$ works so well is that even though the vast majority of balls get inserted using h_1, their contribution to the maximum load is capped at $(1 + o(1))\lambda$ (because, beyond that point, balls are inserted using Greedy$[2]$). This makes it so that the number of balls managed by Greedy$[2]$ at any given moment is only $O(n)$; and therefore the known bound from (6) [49] bounds their contribution to the maximum load as $\log\log n + O(1)$.

We now modify our low-associativity construction to use Iceberg$[2]$ (with $k = 3$ hash functions) instead of just a single hash function. Set the number of bins to be $m/(\log\log P \log\log\log P)$, so that the average load is $\lambda = \log\log P \log\log\log P = \omega(\log\log n)$. Then, with high probability in n (and thus P), the maximum load is

$$(1 + o(1))\lambda + \log\log n + O(1) = \lambda(1 + \delta),$$

with $\delta = o(1)$.[5]

Using this value of δ as the resource-augmentation parameter, it follows that the bin size is (with high probability) at least as large as the maximum load. Since the bin size is $B = \tilde{\Theta}(\log\log P)$, the associativity of the scheme is $3B = \tilde{\Theta}(\log\log P)$. Thus we have a decoupling scheme with huge-page size $h_{\max} = \Theta(w/\log\log\log P)$ and resource augmentation parameter $\delta = o(1)$.

Theorem 3 (The Decoupling Theorem). *There exists a constant-time huge-page decoupling scheme using resource augmentation $\delta = o(1)$ that supports huge-page size $h_{max} = \Theta(w/\log\log\log P)$ with high probability in P.*

Proof. The proof follows just as for Theorem 1, but using Iceberg$[2]$ instead of a single hash function. □

5 OPTIMIZING THE COST OF ADDRESS TRANSLATION

Finally, we consider the task of optimizing the total TLB and IO cost of a memory-management algorithm on a sequence of page requests. More specifically, in this section, we prove that in order to optimize the cost of a memory-management algorithm, it's enough to *independently* optimize the TLB cost and the paging cost, and combine the two solutions via huge-page decoupling. Moreover, these two separate problems are each equivalent to the classic paging problem [47].

We begin by formalizing the definitions of arbitrary memory-management algorithms and of the address-translation cost model. These definitions must carefully address several subtleties of the model. First, what is the full range of control that an *arbitrary* memory-management algorithm has? This needs to be carefully specified so that we we can prove competitiveness results. Second, how do we define the cost of a memory-management algorithm that sometimes brings pages into RAM even when those pages are not being accessed (e.g., a memory-management algorithm that implements virtual huge pages as physical huge pages, and thus brings entire physical huge pages into RAM at once)? And finally, what types of failures are permitted for a memory-management algorithm? In particular, paging failures (as defined in Section 3) are not acceptable, but we shall see that these types of failures can be handled at a cost of additional IOs.

What a memory-management algorithm controls. We begin by extending the definitions from Section 3 in order to define what an arbitrary memory-management algorithm controls. A memory-management algorithm controls:

- which virtual huge-page addresses $\mathcal{T}$ are in the TLB;
- which virtual page addresses are in the active set $\mathcal{A}$;
- what the TLB-decoding function f is;
- and what the virtual-to-physical mapping ϕ is.

In other words, a memory-management algorithm controls not only the features of the system that a huge-page decoupling scheme controls, but also the TLB-replacement policy and the RAM-replacement policy.

Whereas a huge-page decoupling scheme treats $\mathcal{T}$ as consisting of virtual huge pages of size $h_{\max}$, in general, a memory-management algorithm is permitted to use virtual huge pages of any mixture of sizes in $\{1, 2, 4, 8, \ldots, h_{\max}\}$ (we assume $h_{\max}$ is a power of two).[6] Recall from Section 3 that, if a huge page is of size

[3] Interestingly, it is unknown whether the asymptotic dependence on λ is an artifact of the proof. If one could prove a maximum load of $\lambda + O(\log\log n)$ for Greedy$[2]$, then one could use Greedy$[2]$ to achieve the results in this section.

[4] As a minor technical point, the insertions performed using h_1 ignore all balls that were inserted using h_2 and h_3, and, similarly, the Greedy$[2]$ insertion of balls using h_2 and h_3 ignores all balls that were inserted using h_1.

[5] For our purposes here, we do not make an effort to optimize δ beyond ensuring that $\delta = o(1)$. We point out, however, that if one wanted optimize δ further, one could set the associativity to poly$(\log\log P)$ (which only changes $h_{\max}$ by a constant factor), and obtain $\delta = 1/\text{poly}(\log\log P)$, for a polynomial of our choice.

[6] The fact that we allow memory-management algorithms to potentially use many different huge-page sizes at the same time will only make our results stronger. In particular, this will allow the memory-management algorithms $\mathcal{X}$ and $\mathcal{Y}$ that are used as inputs to Theorem 4 to be more sophisticated; on the other hand, the output memory-management algorithm $\mathcal{Z}$ produced by Theorem 4 uses only a single size $h_{\max}$ for huge pages.

2^r, then it is associated with an address that is an integer multiple of 2^r.

Servicing page requests. The purpose of a memory-management algorithm is to service a sequence of virtual-page requests $\sigma = (p_1, p_2, \ldots, p_n)$, where each $p_i \in [V]$.

In order for the memory-management algorithm to be able to service a page request p_i, the algorithm must ensure that virtual page p_i is in RAM (i.e., if $p_i \notin \mathcal{A}$, then p_i must be added to $\mathcal{A}$); and the algorithm must also ensure that a virtual huge page containing p_i is contained in the TLB. Once the page p_i is mapped in both RAM and the TLB, the request p_i can be serviced.[7]

The address-translation cost model. The running time of a memory-management algorithm is evaluated in the ***address-translation cost model***: the cost of adding a new entry to $\mathcal{T}$ is ε and the cost of adding a new element to the active set $\mathcal{A}$ is 1. Evictions (from either the TLB or RAM) are free; and so is updating the TLB value $\psi(u)$ for a virtual huge page address $u \in \mathcal{T}$ when one of u's constituent pages is added or removed from $\mathcal{A}$.

In order to allow for a full range of TLB decoding/encoding schemes, it is necessary to also capture the notion of a ***decoding miss***, which costs ε. A decoding miss occurs if a virtual huge page u is in the TLB, and a virtual page v contained in u is in RAM, but the decoding function $f(v, \psi(u))$ incorrectly evaluates to -1 (instead of $\phi(v)$). Imagine, for example, that a memory-management algorithm chooses to encode for each virtual huge page u in the TLB only the physical addresses of u's most commonly accessed constituent pages; then the pages that do not get encoded would incur decoding misses when they were accessed. We will use decoding misses in Theorem 4 to capture what happens if a huge-page decoupling scheme experiences a paging failure; we will construct a memory-management algorithm that brings the page experiencing failure into RAM without giving it a TLB encoding.

Recall that for a given memory-management algorithm $\mathcal{X}$, $C(\mathcal{X}, \sigma)$ denotes the total cost of $\mathcal{X}$ (on the request sequence σ), $C_{\text{TLB}}(\mathcal{X}, \sigma)$ denotes the total TLB cost (this does not include decoding misses), and $C_{\text{IO}}(\mathcal{X}, \sigma)$ denotes the total IO cost. Additionally, we define $C_{\text{D}}(\mathcal{X}, \sigma)$ as the total cost incurred due to decoding misses. Then, $C(\mathcal{X}, \sigma) = C_{\text{TLB}}(\mathcal{X}, \sigma) + C_{\text{IO}}(\mathcal{X}, \sigma) + C_{\text{D}}(\mathcal{X}, \sigma)$.

Huge-page decoupling as a technique for simultaneously optimizing IO costs and TLB costs. Having defined the address-translation cost model and how it applies to memory-management algorithms, we can now prove Theorem 4.

Theorem 4 (The Simulation Theorem). *Let V and P be the number of pages in the virtual and physical address spaces, respectively. Let ℓ be the number of entries in the TLB, and w be the number of bits in each TLB value.*

Let $\mathcal{D}$ be a huge-page decoupling scheme that uses resource-augmentation $\delta = o(1)$ and supports huge-page size h_{max} with high probability in P.

Let $\sigma = (p_1, \ldots, p_n) \in [V]^n$ be a sequence of virtual page addresses that need to be serviced. Let $\mathcal{X}$ be an arbitrary memory-management algorithm using parameters ℓ, w, V, P, and using huge pages with sizes between 1 and h_{max} pages; let $\mathcal{Y}$ be an arbitrary memory-management algorithm using parameters $\ell, w, V, (1-\delta)P$, and using huge pages with sizes between 1 and h_{max} pages. Using $\mathcal{D}$, one can construct a new memory-management algorithm $\mathcal{Z}$, using virtual huge pages of size h_{max}, satisfying

$$C(\mathcal{Z}, \sigma) \le C_{TLB}(\mathcal{X}, \sigma) + C_{IO}(\mathcal{Y}, \sigma) + \frac{n}{\text{poly}(P)}, \tag{7}$$

with high probability in P. Moreover, if $\mathcal{X}$ and $\mathcal{Y}$ are online algorithms, then so is $\mathcal{Z}$.

Proof. To design $\mathcal{Z}$, we combine all three of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{D}$. The idea is to use $\mathcal{X}$'s TLB-replacement policy, to use $\mathcal{Y}$'s as the RAM-replacement policy, and then to use $\mathcal{D}$ in order to combine those two policies into a new memory-management algorithm $\mathcal{Z}$.

We begin by describing how to use $\mathcal{X}$ to determine the TLB-replacement policy for $\mathcal{D}$. Whereas the TLB for $\mathcal{X}$ may use huge pages of many different sizes (up to size $h_{\max}$), the TLB for $\mathcal{Z}$ uses virtual huge pages exclusively of size $h_{\max}$. Let $\mathcal{T}_{\mathcal{Z}}$ denote the set of virtual huge-page addresses in $\mathcal{Z}$'s TLB at any given moment and let $\mathcal{T}_{\mathcal{X}}$ denote the set of virtual huge-page addresses in $\mathcal{X}$'s TLB at any given moment. For each virtual address $v \in V$, let $r(v) = v - (v \pmod{h_{\max}})$ denote the virtual address of the size-$h_{\max}$ virtual huge page containing v. We say that an address $v \in V$ is ***covered*** by a size-$h_{\max}$ virtual huge-page address u if $r(v) = u$. Note that, if a virtual huge-page address $v \in \mathcal{T}_{\mathcal{X}}$ is covered by a size-$h_{\max}$ virtual huge-page address u, then so are all of the constituent virtual pages v' of v.

We define the TLB-replacement policy for $\mathcal{D}$ so that

$$\mathcal{T}_{\mathcal{Z}} = \{r(v) \mid v \in \mathcal{T}_{\mathcal{X}}\}.$$

Note that, since $|\mathcal{T}_{\mathcal{X}}| \le \ell$, we also always have $|\mathcal{T}_{\mathcal{Z}}| \le \ell$. And, moreover, the TLB-replacement policy only modifies $\mathcal{T}_{\mathcal{Z}}$ if it is also modifying $\mathcal{T}_{\mathcal{X}}$.

Next we describe how to use $\mathcal{Y}$ to determine the RAM-replacement policy for $\mathcal{D}$. We simply have the RAM-replacement policy for $\mathcal{D}$ maintain the active set $\mathcal{A}$ to always match the active set for $\mathcal{Y}$. Note that, since $\mathcal{Y}$ operates on a physical memory of size $(1-\delta)P$, the active set $\mathcal{A}$ is never of size more than $(1-\delta)P$, which in turn meets the resource-augmentation requirement for the huge-page decoupling scheme $\mathcal{D}$.

Although the RAM-replacement policy for $\mathcal{D}$ maintains the active set to match the active set for $\mathcal{Y}$, the huge-page decoupling scheme $\mathcal{D}$ may sometimes experience a paging failure, causing a virtual page v that is in the active set for $\mathcal{Y}$ to not be present in the active set for $\mathcal{Z}$. Whenever a page request p_i is to a page v for which $\mathcal{D}$ is currently experiencing a paging failure, we have the memory-management algorithm $\mathcal{Z}$ handle the page request p_i as follows: (1) the algorithm $\mathcal{Z}$ spends an IO (of cost 1) to temporarily add v to $\mathcal{Z}$'s active set; (2) the algorithm $\mathcal{Z}$ sets $\phi(v)$ to be an arbitrary free physical page address; and (3) the algorithm $\mathcal{Z}$ services the page request to v, incurring an additional cost of ε due to the ensuing

[7] Whereas RAM truly requires that pages be present in order to be accessed, the same requirement isn't strictly necessary for the TLB (since we can always just find the physical address via the page table). On the other hand, in the address-translation cost model, adding an element TLB has the same cost as incurring a TLB miss would. Thus we can assume without loss of generality that every page that is accessed is first added to the TLB's coverage if necessary.

decoding miss (note, in particular, that $\mathcal{Z}$ does not make any effort to encode the translation from v to $\phi(v)$ in the TLB). Thus the total cost of servicing a page request p_i to a page v that is experiencing a paging failure is $1+\varepsilon$. Once the request p_i is serviced, then v may be removed from $\mathcal{A}$ whenever convenient (i.e., whenever $\mathcal{D}$ wishes to assign some other virtual page address $v' \neq v$ to the physical address $\phi(v)$ that v is currently assigned to).

We have now completely defined the memory-management algorithm $\mathcal{Z}$. In summary, $\mathcal{Z}$ is constructed via the huge-page decoupling scheme $\mathcal{D}$ using $\mathcal{X}$ as the TLB-replacement policy and $\mathcal{Y}$ as the RAM-replacement policy; and the only time that $\mathcal{Z}$ departs from the behavior of $\mathcal{D}$ is when $\mathcal{D}$ is experiencing a paging failure on a page request p_i. In this case, $\mathcal{Z}$ serves the page request at a total cost of $1+\varepsilon$. Note that, if $\mathcal{X}$ and $\mathcal{Y}$ are online algorithms, meaning that at any given moment they only know the value of the next page p_i that will be requested, then $\mathcal{Z}$ is also online.

We conclude by analyzing the cost $C(\mathcal{Z}, \sigma)$. First note that, since $\mathcal{D}$ is a huge-page decoupling scheme that succeeds with high probability, the probability that there is a paging failure during a given page request p_i is at most $1/\text{poly}(P)$ (for a polynomial of our choice). By linearity of expectation, the expected number of page requests p_i at which paging failure is being experienced is at most $n/\text{poly}(P)$. Applying Markov's inequality, it follows that with high probability in P, at most $n/\text{poly}(P)$ page requests p_i occur during paging failures. (Note that the application of Markov's inequality shrinks the $\text{poly}(P)$ term in the denominator, but it nonetheless remains a polynomial of our choice.) The total cost incurred by $\mathcal{Z}$ due to paging failures of $\mathcal{D}$ is therefore at most $(1+\varepsilon)n/\text{poly}(P) \leq n/\text{poly}(P)$ with high probability in P.

To complete the proof, we perform the rest of the analysis ignoring costs incurred by $\mathcal{Z}$ due to paging failures. By using $\mathcal{X}$ as the TLB-replacement policy, we ensure that $\mathcal{Z}$ only ever adds elements to $\mathcal{T}_{\mathcal{Z}}$ when $\mathcal{X}$ also adds an element to $\mathcal{T}_{\mathcal{X}}$. Thus (ignoring requests that experience paging failures), the TLB cost of $\mathcal{Z}$ is at most the TLB cost of $\mathcal{X}$. At the same time, by using $\mathcal{Y}$ as the RAM-replacement policy, we ensure that $\mathcal{X}$ only ever adds elements to its active set when $\mathcal{Y}$ adds the same element to its active set (again, ignoring paging failures). Thus the total IO cost of $\mathcal{Z}$ (ignoring paging failures) is at most the total IO cost for $\mathcal{Y}$. Since $\mathcal{Z}$ does not experience any decoding misses except during paging failures, the cost incurred by $\mathcal{Z}$ due to decoding misses is absorbed by the paging failure cost. This completes the proof. □

Theorem 4 reduces the optimization problem of minimizing $C(\mathcal{Z}, \sigma)$ to the independent (and separate) optimization problems of minimizing $C_{\text{TLB}}(\mathcal{X}, \sigma)$ and $C_{\text{IO}}(\mathcal{Y}, \sigma)$. We conclude the section by observing that these two individual optimization problems are equivalent to the classic paging problem, which counts the number cache misses incurred by a paging algorithm to service a sequence of page requests $p_1, p_2, \ldots, p_n$ on a cache of some size. The paging problem does not have a unique optimal online solution (and thus many algorithms for the problem have been studied [11, 12, 19, 20, 52, 53]). Nonetheless the theoretical and practical properties of the paging problem are well understood.

Lemma 1. *Let $\mathcal{X}$ and $\mathcal{Y}$ be the memory-management algorithms from Theorem 4, and let $\sigma = (p_1, p_2, \ldots, p_n) \in [V]^n$. For each virtual page $v \in V$, let $r(v)$ denote the virtual huge page of size h_{max} containing v.*

The problem of minimizing $C_{TLB}(\mathcal{X}, \sigma)$ in Theorem 4 is equivalent to the paging problem on the request sequence $r(p_1), r(p_2), \ldots, r(p_n)$ using a cache of size ℓ.

The problem of minimizing $C_{IO}(\mathcal{Y}, \sigma)$ in Theorem 4 is equivalent to the paging problem on the request sequence $p_1, p_2, \ldots, p_n$ using a cache of size $(1-\delta)P$.

Proof. If we wish to design $\mathcal{X}$ to minimize $C_{\text{TLB}}(\mathcal{X}, \sigma)$, then we can assume without loss of generality that the TLB for $\mathcal{X}$ uses only huge pages of size $h_{\max}$. In particular, any virtual huge page v of size smaller than $h_{\max}$ in the TLB can be substituted with a larger huge page $r(v)$ of size $h_{\max}$ without any increase in TLB cost. If we assume that $\mathcal{X}$ uses huge pages of size $h_{\max}$, then the page request sequence accesses virtual huge pages $r(p_1), r(p_2), \ldots$. We can further assume without loss of generality that $\mathcal{X}$ only adds a virtual huge page v to the TLB when that virtual huge-page is about to be accessed in the request sequence (since otherwise, $\mathcal{X}$ could hold off on adding v until v is next accessed). Thus the problem of minimizing $C_{\text{TLB}}(\mathcal{X}, \sigma)$ is exactly the problem of servicing the virtual huge-page requests $r(p_1), \ldots, r(p_n)$ using the TLB as a size-ℓ cache. Adding a virtual huge page to the TLB in the former problem corresponds exactly to incurring a cache miss in the latter problem.

To see the claim about $C_{\text{IO}}(\mathcal{Y}, \sigma)$, observe that the active set $\mathcal{A}$ for $\mathcal{Y}$ corresponds directly to the cache in the paging problem. Note that, without loss of generality, $\mathcal{Y}$ only adds a page to $\mathcal{A}$ when that page is about to be accessed. Thus the IOs incurred by $\mathcal{Y}$ correspond exactly to the cache misses incurred in the paging problem. □

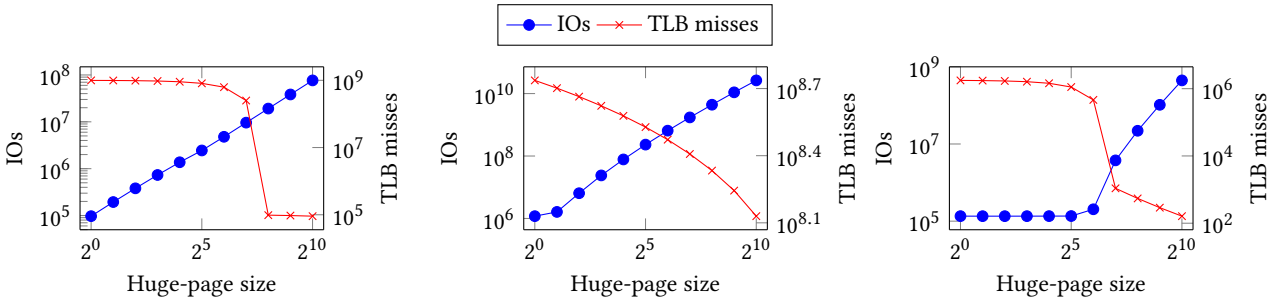
6 THE IO TLB-MISS TRADE-OFF IN HUGE PAGES

This section presents experimental data illustrating the trade-off between IOs and TLB misses when using huge pages (that are both virtually and physically contiguous).

We developed a trace-driven simulator for the TLB and RAM and used it to measure IOs and TLB misses as a function of the huge-page size $h \in \{1, 2, 4, \ldots, 1024\}$. We use a base page size of 4kB. Thus, each entry in the TLB represents h 4kB virtually contiguous pages, which map to an equal number of physically contiguous pages. Therefore, each page fault moves h pages between RAM and secondary memory, at a cost of h IOs. For our experiments, we regard the TLB as a fully associative cache and use LRU as the replacement policy both for the TLB and RAM. For all experiments, the TLB is modeled with 1536 entries. The amount of physical memory modeled varies, as detailed below.

We ran the following workloads:

- **Bimodal uniform accesses (Figure 1a):** A synthetic stress test that frequently accesses one "hot" page and infrequently accesses another "cold" page. The "hot" page is selected at random from a 1 GB region of memory, within a 64 GB virtual address space; the "cold" page is selected at random from the entire virtual address space. This workload is designed to be a worst case for huge pages. Small h results in frequent TLB



(a) A bimodal distribution in which 99.99% of accesses are uniformly random in a 1GB working set, and the remaining accesses are uniformly random over a 64GB virtual-address space. Cache was 16GB.

(b) A random walk among pages, where each page has a logarithmic number of outgoing edges, and edge destinations are Pareto distributed with Pareto parameter $\alpha = 0.01$, i.e. the probability that an edge goes to the ith page is proportional to $i^{-\alpha-1}$. Cache was 32GB.

(c) graph500, a BFS on a large graph with a 520MB cache.

Figure 1: IOs and TLB misses as a function of the huge-page size for a bimodal uniform random workload, a random graph walk, and for a trace from the graph500 benchmark. All TLBs had 1536 entries. In all workloads, increasing the huge-page size increases the IO cost by at least three orders of magnitude, but reduces the TLB miss count by up to four orders of magnitude.

misses on accesses to the 1GB virtual region, whereas large h incurs large IO amplification on the infrequent 64GB space accesses. The size of RAM is 16GB. We performed 100 million accesses to warm up the cache, then measured IOs and TLB misses for another 100 million accesses.

- **Random walk on a graph (Figure 1b):** A synthetic workload that performs a random walk on a large graph, modeling a PageRank-like computation. We model each page as a node in the graph, where each node has a logarithmic number of outgoing edges. The destination page of each outgoing edge is chosen from a Pareto distribution over all the pages in the system, with Pareto constant $\alpha = 0.01$ (i.e., the probability of selecting the ith page is proportional to $i^{-\alpha-1}$). The size of the allocated virtual memory is 64GB and the cache has size 32GB. We performed 100 million warm-up accesses, and then measured TLB misses and IOs for 100 million more accesses.
- **graph500 (Figure 1c):** A well-known data-intensive high-performance computing benchmark [31] that performs a BFS traversal on a large graph. We ran the simulator on a trace that consists of approximately 5 million memory accesses performed by graph500 during a period of high memory pressure and high TLB miss rate. The trace was recorded from an execution on a machine with 64GB of RAM, and the memory footprint was 60GB. During this trace, graph500 touches roughly 525MB of RAM. Our simulator's RAM was set slightly below this value, at 520MB, to create some memory contention.

Notice that, in all three workloads, if we don't use huge pages at all (i.e., the huge page size is 1), then the TLB miss count is 1 to 4 orders of magnitude larger than the IO count. Thus these are the type of workloads where huge pages can help TLB performance.

All three workloads exhibit a similar trend: On the one hand, without huge pages, there are relatively few IOs, but a relatively large number of TLB misses. On the other hand, if we use large huge pages, then the TLB misses plummet, but the workloads incur several orders of magnitude more IOs. There is no good choice for the huge page size that simultaneously attains low IO cost and low TLB miss count—huge pages can be a boon for TLBs but a bane in terms of IO.

These experiments show that physically contiguous huge pages have an unsavory trade-off between TLB misses and IO costs. In contrast, a huge-page decoupling scheme has the potential to realize both the low TLB miss rates of huge pages while retaining the low IO costs of regular-sized pages.

7 RELATED WORK

Limited-associativity paging. Sleator and Tarjan [47] gave competitive analyses of LRU and FIFO paging algorithms, both with and without resource augmentation. These results were subsequently generalized by many authors [11, 12, 19, 20, 52, 53].

Due to the importance of limited-associativity caches in hardware, there has also been substantial theoretical work on paging algorithms in the low-associativity setting. One direction of work has been to analyze the competitive ratio of low-associativity paging algorithms, where OPT is *also* limited in its associativity [6, 13, 14, 23, 39]. Another direction of work has been to design cache-aware algorithms that interact well with caches of low associativity. Notably, Frigo et al. [24, 25] and Prokop [43] showed how to take any algorithm in the external-memory model [5] and change the algorithm's access patterns in order so that a direct-mapped cache (i.e., a cache with associativity 1) can be used to simulate a fully-associative cache up to a constant factor in performance. In a similar direction, Sen and Chatterjee [46] present cache-aware

algorithms for several basic problems (e.g. sorting, FFT, and permutations) in a variant of the external-memory model [5, 24, 25] in which cache has limited associativity.

In contrast with past work, our results show how to convert *any* fully associative paging scheme into one with limited associativity at *almost no* overhead. Thus, rather than designing a paging algorithm (or designing an algorithm whose memory accesses play well with a paging algorithm), we are interested deciding *where* pages should reside in memory. And rather than aiming for a constant competitive ratio, our application of address translation requires us to be $(1 + o(1))$-competitive with the paging algorithm that we are simulating. On the other hand, whereas past work often treats the associativity as constant (and possibly even 1), our schemes are allowed to use super-constant associativity (although, remarkably, we show that even $\tilde{O}(\log \log P)$-associativity suffices).

Huge pages. Increasing the granularity of address translation is a standard method to amplify TLB coverage. For instance, Linux provides software support for huge pages of size larger than the typical 4kB [26]. Manufacturers typically manage heterogeneity of page sizes using dedicated TLBs for different sizes. For instance, Intel's Cascade Lake microarchitecture allows 2MB and 1GB pages, and provides a 1536-entry L2 data TLB for 4kB and 2MB pages, and a 16-entry L2 data TLB for 1GB pages [15]. The actual coverage gains are limited by the dedicated TLB size, and are thus much less than the multiplicative blowup in page size.

For some workloads, huge pages are wasteful, creating unnecessary memory pressure that is in turn worsened by the increased swapping cost. Two attempts to overcome this lack of flexibility are Linux's *transparent huge pages* (THP) and *superpages* [32], that work by coalescing areas of virtually and physically contiguous pages into larger blocks (a huge/super page). In these schemes, the OS must either enforce contiguity of physical huge pages, or have fallback mechanisms when it cannot allocate a physical huge page. THP attempts to reserve enough space for a huge page and, in case of failure, falls back to allocating typical 4kB pages that are reallocated later on. Page reallocation incurs large performance penalties, since all applications whose pages are being moved are paused, and in fact a number of commercial databases and other products recommend huge pages should be disabled for optimal performance [1–4]. The superpage system avoids reallocation by always over-allocating memory, and keeps track of unused pages within a superpage so they can be reclaimed by other superpages. The downside is an increased complexity and overhead of OS memory management. Both approaches suffer from increased swapping costs, because once a huge or superpage is created, it is treated as an indivisible mapping unit.

For some workloads huge pages increase page fault latency because Linux has to clear up much larger pages and consolidates fragmented pages to create large continuous physical pages synchronously. Ingens [30] points out that workloads that fragment memory quickly, such as in multi-tenant cloud environments, suffer significant performance penalty, and implements an adaptive policy to promote huge pages and asynchronously defragment memory. HawkEye [35] proposes to synchronously pre-zero freed pages to further reduce latency. GLUE [42] observes that huge pages hurt lightweight system memory management and reduce consolidation in the over-committed cloud deployments which depends on page sharing to share memory. TEMPO [10], a prefetching optimization technique to reduce TLB misses, reports in experiments how much huge pages de-optimize: the more frequent huge pages are used the less effective the optimization becomes.

TLB encodings. Because of the difficulties in maintaining physical contiguity, a number of research projects have explored practical TLB optimizations that leverage some contiguity when it is present, such as coalescing TLB entries for runs of contiguous translations that are smaller than a huge page [17, 38, 40, 41], or composing a huge page out of "medium" sized frames [21]. Direct Segments [8] allow a programmer to map gigabyte- to terabyte-sized primary segments of memory and making the hardware to represent these segments using a single TLB translation entry. Proposals such as COLT [41] and Translation Ranger [50] identify physically contiguous pages mapped in a process address space, and compress the TLB translation into a single entry. As these examples show, huge paging schemes that require physical contiguity saddle the OS developer with solving the difficult, open problem of efficiently maintaining physical contiguity.

To the best of our knowledge, our work is the first to completely remove the requirement that huge pages be stored physically contiguously.

8 CONCLUSION

In addition to showing how to potentially improve address translation on existing hardware, our results suggest ways that it may make sense to change hardware in the future to improve address translation.

Specifically, this paper treats w as a fixed parameter. On the other hand, when designing the hardware of TLBs, there is an opportunity to change the value of w if the payoff is big enough. An interesting feature of our results is that they change the asymptotic relationship between w and the coverage of the TLB: even small increases in w correspond to potentially large gains in TLB coverage (and, moreover, these gains do not require the storage of additional keys!). Thus larger values of w may make sense using our techniques than was previously the case.

On the other hand, as long as w remains reasonably small, then a hybrid approach may be sensible: one can use both huge-page decoupling and physical huge pages of moderate size. So, for example, if an optimal virtual huge page size is $q \gg h_{\max}$ pages, then we could implement decoupled huge pages where the *physical* huge pages would have size only $q/h_{\max}$, thus achieving all the coverage of the very large huge pages while mitigating the adverse effects on I/Os.

9 ACKNOWLEDGEMENTS

This research was supported in part by NSF grants CCF-2106827, CCF-1725543, CSR-1763680, CCF-1716252, CNS-1938709, CCF-1617618, CCF-1916817, CCF-2106999, CSR-1938180 and CCF-1715777, as well as an NSF GRFP fellowship and a Fannie and John Hertz Fellowship.

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

REFERENCES

[1] Couchbase: Disabling transparent huge pages (THP). https://docs.couchbase.com/server/current/install/thp-disable.html. Accessed: 2/11/2021.
[2] MongoDB: Disable transparent huge pages (THP). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/. Accessed: 2/11/2021.
[3] Oracle database: Disabling transparent hugepages. https://docs.oracle.com/en/database/oracle/oracle-database/12.2/ladbi/disabling-transparent-hugepages.html. Accessed: 2/11/2021.
[4] Percona: Settling the myth of transparent hugepages for databases. https://www.percona.com/blog/2019/03/06/settling-the-myth-of-transparent-hugepages-for-databases/. Accessed: 2/11/2021.
[5] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.
[6] Kunal Agrawal, Michael A. Bender, and Jeremy T. Fineman. The worst page-replacement policy. In *Proceedings of the 4th International Conference on Fun with Algorithms (FUN)*, page 135–145. Springer-Verlag, 2007.
[7] Inc. AMD. Amd-v nested paging.
[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2013.
[9] Abhishek Bhattacharjee. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro*, 37(5):6–10, 2017.
[10] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–76. ACM, 2017.
[11] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, USA, 1998.
[12] Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. The relative worst-order ratio applied to paging. *J. Comput. Syst. Sci.*, 73(5):818–843, August 2007.
[13] Mark Brehob, Richard Enbody, Eric Torng, and Stephen Wagner. On-line restricted caching. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 374–383. Society for Industrial and Applied Mathematics, 2001.
[14] Niv Buchbinder, Shahar Chen, and Joseph (Seffi) Naor. Competitive algorithms for restricted caching and matroid caching. In *Proceedings of the 22nd European Symposium on Algorithms (ESA)*, pages 209–221. Springer Berlin Heidelberg, 2014.
[15] Intel's Cascade Lake microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. Accessed: 02/02/2020.
[16] Fernando J. Corbató. A paging experiment with the Multics system. In *MIT Project MAC Report MAC-M-384*, 1969.
[17] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 435–448. ACM, 2017.
[18] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.
[19] Reza Dorrigiv and Alejandro López-Ortiz. Closing the gap between theory and practice: New measures for on-line algorithm analysis. In Shin-ichi Nakano and Md. Saidur Rahman, editors, *WALCOM: Algorithms and Computation*, pages 13–24. Springer Berlin Heidelberg, 2008.
[20] Reza Dorrigiv, Alejandro López-Ortiz, and J. Ian Munro. On the relative dominance of paging algorithms. *Theor. Comput. Sci.*, 410(38–40):3694–3701, September 2009.
[21] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234, Feb 2015.
[22] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685 – 699, 1991.
[23] Amos Fiat, Manor Mendel, and Steven Seiden. Online companion caching. *Theoretical Computer Science*, 324:499–511, 09 2002.
[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, page 285, 1999.
[25] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.
[26] Mel Gorman. Linux huge pages. https://lwn.net/Articles/375096/, 2010.
[27] Mel Gorman. AMD Zen architecture. https://en.wikichip.org/wiki/amd/microarchitectures/zen, 2018.
[28] Inc. Intel. Intel® 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1.
[29] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, 2016.
[30] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 705–721. USENIX Association, November 2016.
[31] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.
[32] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating System Design and Implementation (OSDI)*, 2002.
[33] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. *CoRR*, abs/1902.02411, 2019.
[34] Omitted for Anonymity. Dynamic balls-and-bins and iceberg hashing. Under review, 2021. Manuscript available upon request.
[35] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–360. Association for Computing Machinery, 2019.
[36] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. *SIGPLAN Not.*, 53(2):679–692, March 2018.
[37] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. *SIGARCH Comput. Archit. News*, 45(2):444–456, June 2017.
[38] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 444–456. Association for Computing Machinery, 2017.
[39] Enoch Peserico. Online paging with arbitrary associativity. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564. Society for Industrial and Applied Mathematics, 2003.
[40] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, Feb 2014.
[41] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: coalesced large-reach TLBs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–269, December 2012.
[42] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Dec 2015.
[43] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.
[44] Martin Raab and Angelika Steger. "balls into bins" — a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer Berlin Heidelberg, 1998.
[45] SandyBridge. https://www.7-cpu.com/cpu/SandyBridge.html.
[46] Sandeep Sen and Siddhartha Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 829–838, USA, 2000. Society for Industrial and Applied Mathematics.
[47] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.
[48] Michael M. Swift. Towards $O(1)$ memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 7–11, 2017.

[49] Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, July 2003.
[50] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 698–710, New York, NY, USA, 2019.
[51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 111–125, Santa Clara, CA, February 2020. USENIX Association.
[52] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, Jun 1994.
[53] Neal E. Young. On-line file caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 82–86, USA, 1998. Society for Industrial and Applied Mathematics.
[54] AMD's Zen microarchitecture. https://en.wikichip.org/wiki/amd/microarchitectures/zen. Accessed: 07/15/2020.