MIT Libraries | DSpace@MIT

MIT Open Access Articles

Mosaic Pages: Big TLB Reach with Small Pages

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Gosakan, Krishnan, Han, Jaehyun, Kuszmaul, William, Mubarek, Ibrahim, Mukherjee, Nirjhar et al. 2023. "Mosaic Pages: Big TLB Reach with Small Pages."

As Published: https://doi.org/10.1145/3582016.3582021

Publisher: ACM|Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3

Persistent URL: https://hdl.handle.net/1721.1/150362

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

Mosaic Pages: Big TLB Reach with Small Pages

Krishnan Gosakan[*]
Rutgers University
USA
krishnan.gosakan@gmail.com

Jaehyun Han[*]
The University of North Carolina at Chapel Hill
USA
jaehyun@cs.unc.edu

William Kuszmaul
Massachusetts Institute of Technology
USA
kuszmaul@mit.edu

Ibrahim N. Mubarek
Carnegie Mellon University
USA
imubarek@andrew.cmu.edu

Nirjhar Mukherjee
Carnegie Mellon University
USA
nirjhar@cmu.edu

Karthik Sriram
Yale University
USA
karthik.sriram@yale.edu

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

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

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

Abhishek Bhattacharjee
Yale University
USA
abhishek@cs.yale.edu

Alex Conway
VMware Research
USA
me@ajhconway.com

Martin Farach-Colton
Rutgers University
USA
martin@farach-colton.com

Jayneel Gandhi
Meta
USA
jayneel@meta.com

Rob Johnson
VMware Research
USA
robj@vmware.com

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

Donald E. Porter
The University of North Carolina at Chapel Hill
USA
porter@cs.unc.edu

ABSTRACT

The TLB is increasingly a bottleneck for big data applications. In most designs, the number of TLB entries are highly constrained by latency requirements, and growing much more slowly than the working sets of applications. Many solutions to this problem, such as huge pages, perforated pages, or TLB coalescing, rely on physical contiguity for performance gains, yet the cost of defragmenting memory can easily nullify these gains.

This paper introduces mosaic pages, which increase TLB reach by compressing multiple, discrete translations into one TLB entry. Mosaic leverages virtual contiguity for locality, but does not use physical contiguity. Mosaic relies on recent advances in hashing theory to constrain memory mappings, in order to realize this physical address compression without reducing memory utilization or increasing swapping.

This paper presents a full-system prototype of Mosaic, in gem5 and modified Linux. In simulation and with comparable hardware to a traditional design, mosaic reduces TLB misses in several workloads by 6–81%. Our results show that Mosaic's constraints on memory mappings do not harm performance, we never see conflicts before memory is 98% full in our experiments — at which point, a traditional design would also likely swap. Once memory is over-committed, Mosaic swaps fewer pages than Linux in most cases. Finally, we present timing and area analysis for a verilog implementation of the hashing function required on the critical path for the TLB, and show that on a commercial 28nm CMOS process; the circuit runs at a maximum frequency of 4 GHz, indicating that a mosaic TLB is unlikely to affect clock frequency.

[*]Both authors contributed equally

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582021

CCS CONCEPTS

• **Software and its engineering** → **Virtual memory**; **Main memory**; • **Computer systems organization** → **Architectures**.

KEYWORDS

virtual memory, address translation, TLB, paging, hashing

ACM Reference Format:
Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582021

1 INTRODUCTION

Data-hungry applications, such as data and graph analytics, are often bottlenecked on the translation lookaside buffer (TLB). A typical TLB can only cache a relatively small number of address translations—often caching fewer translations than the working sets of these applications. For example, the data-intensive Graph500 benchmark, when running a breadth first search on a tree with over 2^{20} nodes, has an approximate working set size of 215 MiB, whereas a typical TLB using 4 KiB pages can only cache translations for about 8.6 MiB of physical memory at once. Some of these applications, such as graph analytics, also tend to have irregular, pointer-based memory traversals with poor locality of reference, thwarting common heuristics such as prefetching. As a result, many modern applications report 20–30% overhead attributable to TLB misses [19, 20, 32, 54], and some as high as 83% [5].

Address translation has become a bottleneck because TLBs have few entries, and their growth is much slower than system memory capacity growth. For instance, upcoming disaggregated memory technologies have increased main memory into the terabytes [35], yet Intel's recent Golden Cove server chips have only 2,488 discrete translations across all page sizes, even in the larger, L2 TLB [14]. TLBs have few entries to satisfy architectural constraints. L1 TLBs are on the critical path for L1 cache accesses, and therefore must be extremely fast. Moreover, TLBs tend to be power-hungry, consuming 3-13% of a processor's power, and higher associativity and deeper TLB hierarchies further increase dynamic energy usage [29].

One of the main techniques for increasing TLB reach is to increase the size of translation units, via the use of ***huge pages***, segments, or opportunistic coalescing of contiguous entries. These techniques depend upon physical contiguity, and therefore incur the costs of defragmentation. Defragmenting physical memory is expensive and has no good solutions in the worst case—so much so that defragmentation can overwhelm any performance gains from greater TLB reach. For instance, Zhu et al. [66] recently report that a cold cache Redis workload shows a 29% throughput gain on Linux when switching from 4 KiB pages to transparent 2 MiB pages—with no fragmentation; when memory is 50% fragmented on Linux, however, throughput with 2 MiB pages drops to only 89% of the throughput with 4 KiB pages.

Valid?	Tag (Virtual Page #)	Physical Frame #
1	0x1010	
1	0x1011	
1	0x1012	
1	0x1013	
0	...	...

Valid?	Tag (Mosaic Virtual Pg #)	Compressed Physical Frame Numbers (CPFNs)
1	0x101	
1	0x138	
0	...	...

Figure 1: The top depicts a stylized, traditional TLB, which maps virtual addresses (tags) to physical page frames. In this figure, a series of four virtually contiguous pages map to different physical addresses. The bottom shows how a Mosaic TLB compresses the same run of physical pages into one entry, by only storing the bucket and offset for each page. We note that the figures are truncated for brevity, but we assume a similar cache geometry for a traditional or Mosaic design.

Finally, some proposals mitigate fragmentation in large pages by adding more complex hardware, which can stitch together physically discontiguous sub-pages into a larger huge page [16, 53, 65]. For instance, perforated pages [40] introduce a shadow page table layer that redirects portions of a huge page to 4 KiB pages without the need to defragment. A key point about these designs is that the performance gains still come from the residual physical contiguity in the mappings; filling holes and combining sub-mappings is arguably more efficient in total than defragmenting memory.

This paper introduces ***mosaic pages***, a technique for increasing TLB reach **without using physical contiguity**. Without the need for physical contiguity, one need not defragment memory. To demonstrate the feasibility and capabilities of ***mosaic pages***, we present **Mosaic**, an *end-to-end redesign* of address translation mechanisms across the hardware TLBs and the OS. Mosaic internally uses the recently developed Iceberg hashing [8] for physical address compression and mitigating TLB conflicts.

Physical address compression. The key idea behind mosaic pages is to compress each address translation, so that multiple, virtually contiguous translations fit into a single TLB entry, illustrated in Figure 1. We achieve our compression by restricting each virtual address to map to only a small number h of physical page frames (via hashing), so that a virtual page's physical address can be encoded using only $\log h$ bits. For concreteness, we set $h = 104$ in our experiments, which means we encode each translation in seven bits. In contrast, conventional virtual memory systems allow each virtual page to be mapped to (almost) any of the p physical page frames, requiring $\log p$ bits per address. We call one discrete translation a ***Compressed Physical Frame Number (CPFN)***, encoding which of the h page frames this virtual address maps to.

By compressing translations, we can pack translations for several contiguous virtual pages into a single TLB entry, expanding TLB reach by $a = \log p / \log h$ without increasing the number TLB entries. As we show in Section 3.1, we can increase coverage by at least $a = 4$ using TLB entries of the same size as in today's hardware. By widening TLB entries, we can plausibly increase a to 64 without prohibitive costs.

Like huge pages, mosaic pages leverage virtual contiguity but, unlike huge pages, do not require physical contiguity. In our design, each TLB sub-entry can be mapped independently. Furthermore, mosaic pages compose with other techniques, such as huge pages, because any base page size can be mapped by TLB sub-entries.

Mitigating conflicts. The concern with reducing h is that it increases ***conflicts*** in mapping virtual addresses to physical pages, and resolving these conflicts has a cost. Specifically, when mapping a new virtual page, we may find that its h allowed locations are already occupied by hot pages. In this case, the conflict must be resolved, e.g., by swapping a conflicting page to disk. Mapping restrictions may force the eviction of a hotter page than an unconstrained mapping would. The more restricted the mapping (i.e., the smaller h), the more likely it is that a conflict will make a worse eviction choice than an unconstrained mapping. On the other hand, smaller h decreases the size of TLB encodings, which mosaic uses to increase TLB reach. One of the principal contributions of this paper is showing that **it is possible to have a small h with comparable swapping costs.**

Mosaic overview. Mosaic structures physical memory as a bucketed hash table, where each bucket consists of a collection of contiguous physical page frames, which we call ***slots***. Each virtual address is hashed to a small number d of the buckets, and the virtual page may be placed in any of the h total slots among those d buckets. Thus, the TLB only needs to store an encoding of which of the h slots was chosen. In our experiments each virtual address is mapped to one bucket of 56 slots and 6 buckets of 8 slots, for a total of $h = 104$ slots, so the TLB entry effectively encodes a (bucket, slot) tuple as a small integer (§2.2 and §3.1).

We note that although this paper uses hashing to increase TLB reach, mosaic does not require hashed or inverted page tables. A mosaic page table should support looking up the CPFN (instead of PFN) for a virtual address, but can otherwise be structured using radix trees, hash tables, etc. Our prototype uses a modified version of the standard radix-tree page table (§3.1).

Mosaic's hashing scheme. The hash table underlying mosaic's page allocation should have the following properties: it must allocate pages to a small number of places (in order to reduce the size of a CPFN), it must, with very high probability, operate successfully at load factors within a few percent of 100% (so that each slot can be usefully occupied), and it must be ***stable***, meaning that it does not move items to resolve conflicts (to avoid complex and expensive page migration).

Interestingly, these properties seem to be at odds with each other, and it's not obvious that they can be attained simultaneously. For instance, one way to realize high load factors is to migrate entries in the table, as in cuckoo hashing, but this violates the stability goal. In fact, Iceberg hashing [7, 8, 38], the hashing scheme that we use in Mosaic, was first proposed only last year and is the first hash table to provably meet all of these criteria. Section 2.3 explains how iceberg hashing obtains all these seemingly contradictory properties simultaneously.

Our contributions. This paper contributes an end-to-end system co-design and implementation of mosaic pages, from the architecture to the OS. We implement the TLB changes in the gem5 simulator [12], and modify Linux to implement mosaic for anonymous, unshared pages.

Using this experimental infrastructure, we give a thorough demonstration that mosaic can indeed reduce TLB misses of real-world workloads, such as Graph500, by 6–81% in simulation with comparable TLB entry width as a current x86 chip.

Second, we contribute an implementation of our hashing scheme for the TLB in Verilog and measure it on an FPGA and with a 28nm commercial CMOS process. The timing analysis yields a maximum clock frequency of 4 GHz, indicating that the hashing we add to the critical path is unlikely to harm overall clock frequency or have significant area cost.

Finally, the paper demonstrates empirically that mosaic under memory pressure has swapping comparable to an unconstrained page mapping. We measure swapping events on longer, bare metal workloads under two conditions: sufficient memory and insufficient memory. Our experiments show that, commensurate with Iceberg's probabilistic bounds, as long as only 2% of memory is held in reserve and the application(s) fit into DRAM, conflicts are not observed. We find that the system swaps only after memory is over 98% utilized—similar to unmodified Linux swapping once memory is fully utilized. Once memory is over-subscribed, mosaic typically swaps *less* than default Linux.

Future work. The mosaic prototype leaves some features for future work. Most notably, we do not demonstrate support for shared memory mappings in the iceberg design, although we do outline how one might add this feature in future work (§2.5). More broadly, modern memory protection has incorporated a number of features, such as encryption (Intel TME, AMD SEV), sub-page protections (e.g., Intel MPK), trusted execution environments (e.g., Intel's SGX), and nested paging; a production deployment of mosaic would need to integrate with a very long tail of features. This paper argues that the performance gains of mosaic are sufficiently appealing to warrant this future work.

2 MOSAIC PAGES

In this section, we describe the design and theoretical background for mosaic pages.

2.1 Overview of Mosaic Pages

A ***mosaic page*** is a large virtual page, composed of a virtually consecutive, but not necessarily physically contiguous, base pages (4 KiB). We say that a is the ***arity*** of a mosaic page. For concreteness, we use $a = 4$ as a default setting, but experiment with powers of two up to 64 in the evaluation (§4). Moreover, it is even possible

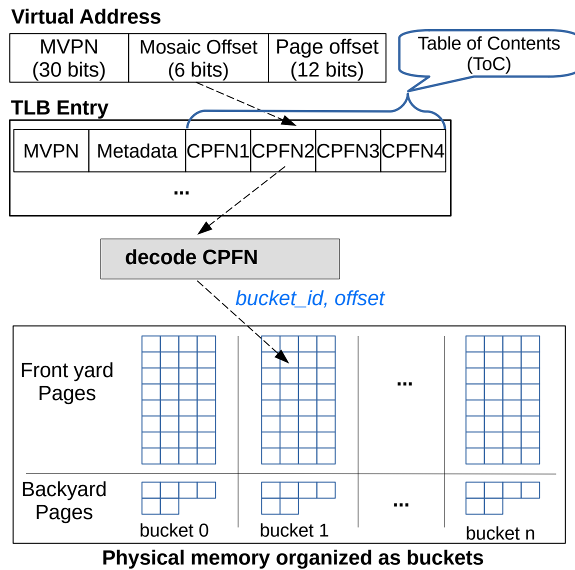


Figure 2: High-level Design of Mosaic Address Translation. Physical memory is organized as buckets in a hash table; buckets have a front and back yard for load balancing (§2.3). The TLB is indexed by the upper bits of the virtual address (the Mosaic virtual page number), and the TLB entry stores a run of $a = 4$ compressed physical addresses (CPFNs), which are virtually but not physically contiguous. Compression is realized through hashing a virtual address to a small number of physical frames; the TLB need only store which of the small number of frames was chosen—here, a bucket and offset.

that only a subset of the pages are in memory at any given moment. Hence, mosaic avoids the complexities and costs of maintaining physical contiguity.

The key idea is to compress each translation such that translations for all $a = 4$ base pages fit in one TLB entry, as illustrated in Figure 2. Although the frames are allocated independently, we will ensure that each page's location can be encoded with just a few bits of information—these bits are known as the ***compressed physical frame number*** (CPFN) of the page. The TLB is indexed by ***mosaic virtual page number (MVPN)*** (or the aligned, virtual address of the mosaic page), and each entry in the TLB holds a series of CPFNs for each virtual page in that mosaic page. Together, we call these CPFNs the ***table of contents (ToC)*** for the mosaic page.

A TLB lookup for a virtual address returns the ToC for the relevant mosaic page. The base page offset within the mosaic page (or ***mosaic offset***) then determines which entry in the *ToC* corresponds to the desired virtual page. The CPU then uses the CPFN to compute the page's actual page frame number (PFN), which is explained in §2.2.

Mosaic pages increase the reach of the TLB by a factor of a by leveraging virtual locality, without requiring physical contiguity. To get a ballpark estimate of how much we can increase TLB reach, we consider current x86 TLBs, which use 36-bit physical frame numbers. If we use 8-bit CPFNs, then we can fit $a = 4$ CPFNs in a single TLB entry, increasing TLB reach by a factor of 4. Furthermore, there is good reason to believe that we can actually increase the width of TLB entries without incurring too much cost in terms of power or chip area, so a future production implementation might have $a = 16$ or even larger.

TLBs typically store several pieces of metadata about each page, such as permissions and accessed/dirty bits. In our prototype, we store valid bits per CPFN, but assume other bits can be tracked at the granularity of a mosaic page, similar to other recent work on extending TLB reach [5, 45]. We expect that most applications could be recompiled with linker directives to ensure all mappings are created with similar alignment and permissions. One could also widen the TLB and store all of these entries on a per-base-page basis.

Mosaic page tables map MVPNs to ToCs, but mosaic can use any page-table structure, such as radix trees, hash tables, or even a software-managed TLB, as long as they don't impose any page mapping restrictions that conflict with mosaic's, described next.

2.2 Compressed Physical Frame Numbers

The key to compressing PFNs is that whenever we need to allocate a physical frame for virtual address v, we limit ourselves to a small set of possible frames (h; for concreteness, $h = 104$ in our experiments). We use the term **associativity** to describe these limits on the number of frames that can map a given 4 KiB base page; although associativity is a common concept in hardware caches, here we are not discussing how data can be placed in CPU caches, but as a restriction on virtual page mappings. Thus the CPFN needs to indicate only which of the h options was chosen by the page allocator. This means that a CPFN can be stored simply as a number in the set $\{0, 1, , \ldots, h-1\}$, which requires only $\log h$ bits.

Our page allocator treats the frames in physical memory as slots in a hash table, in which slots are grouped into buckets. Each VPN is mapped to one or more buckets via a hashing scheme, and the CPFN records which bucket and which slot within that bucket were chosen by the allocator. In practice, we hash (ASID, VPN) pairs, but for simplicity we omit the ASID except when relevant.

Note that this contrasts with conventional virtual memory schemes, in which every virtual page can be mapped to any physical frame. Thus, conventional virtual memory schemes are ***fully associative***, whereas mosaic is a low-associativity virtual-memory scheme.

The potential drawback of low associativity is that, if it is implemented naively, then *associativity conflicts* can become a problem. That is, because each virtual address has only h options for where it can go in physical memory, the system may be forced to swap some other virtual address that we would not otherwise choose to swap out. Thus, even if our working set size is smaller than physical memory, associativity conflicts might prevent mapping the entire working set at once.

There are two core details to be filled in to complete this design: what hashing scheme do we use for page allocation, and what swapping algorithm to use?

2.3 Low-Associativity Page Allocation with Hashing

The hashing scheme we use in our page allocation scheme must meet three criteria:

(1) **Low Hashing Associativity:** For each item (i.e., virtual address) the set of possible positions where the item could reside in the hash table is less than or equal to a small h.
(2) **Stability:** Once an item is inserted into the hash table, it is not moved until a future deletion removes it. This implies that once mapped, pages never need to be copied within memory to ensure good performance, whereas schemes like cuckooing must migrate elements to maintain performance.
(3) **High Utilization:** If p is the total number of slots in the hash table (i.e., the total number of physical frames), then the hash table can handle up to $(1-\delta)p$ elements at a time for some small δ ($\delta \approx .02$ in our experiments). Practically speaking, this means that nearly all of memory can be allocated (98% in our experiments) before seeing conflicts, with extremely high probability.

Iceberg hashing. Mosaic allocates pages by using ***Iceberg hashing*** [8], a recently proposed hashing scheme that achieves the above three criteria simultaneously, which had long been an open problem in hash-table design.

Many classical hash tables meet two of the three. For example, cuckoo hashing has low associativity and can have a load factor of > 90%, but moves items around (i.e., it is unstable). Other open-addressing schemes such as linear probing or quadratic probing can be stable and support high load factors, but have high associativity.

We now review Iceberg hashing, since it will be relevant to the overall design of mosaic. An Iceberg hash table consists of two components: a ***front yard*** and a (much smaller) ***backyard***, illustrated in Figure 2. The front yard is broken into s bins of some fixed size $f = \omega(\log\log p)$ (e.g., $f = \Theta(\log^2 \log p)$). The backyard also consists of s bins, each with capacity $b = \Theta(\log\log p)$, where p is the total number of slots in the hash table (i.e., the total number of frames in physical memory). Note that front yard and backyard buckets can be quite small in practice. For example, for 64-bit systems, $\log\log p \approx 5.7$, so a reasonable choice would be front yard buckets of size $5.7^2 \approx 32$ (or larger) and backyard buckets of size ≈ 5.7 (or larger).

We illustrate insertion in Figure 3. Whenever an item x is inserted, it first hashes to some bin $h_0(x)$ in the front yard. If there is a free slot in $h_0(x)$, then the insertion uses that slot (as illustrated in the first case of Figure 3). Otherwise, if bin $h_0(x)$ is full, then x is placed into the backyard. Elements in the backyard are assigned a bin using the power of d choices: the element hashes to d bins $h_1(x), \ldots, h_d(x)$ and is placed in the emptiest of those bins. This is illustrated in the second case of Figure 3.

The full dynamics of the front yard/backyard scheme are quite difficult to analyze [8, 9]. What one can show, however, is that as long as $f = \omega(\log\log p)$, then the number of elements in the backyard will always be $o(p/\log\log p)$. Then, a classic theoretical result on the power-of-d-choices [59] guarantees that the probability of bins in the backyard overflowing is negligible, so the hash table supports space utilization $1-\delta$ for very small $\delta = o(1)$. These

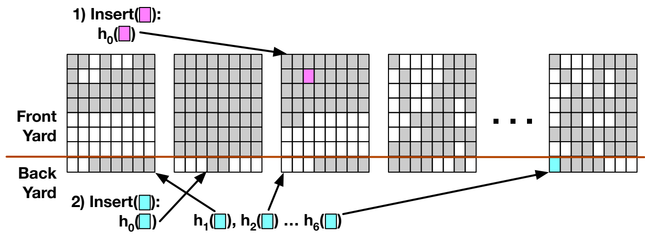


Figure 3: Allocation of pages in mosaic, where physical memory is modeled as an Iceberg hash table. Pages are divided into buckets, split into a front yard and backyard. For specificity, we illustrate using the mosaic prototype parameters: bucket size of 64 page frames, split into a front yard of 56 frames and a backyard of 8 frames. Gray frames are in use. The first insertion of the fucshia page mapping uses h_0 to select a front yard bucket; in this case, there is space so the mapping is created. In the second insertion of the cyan page mapping, the front yard bucket is full. Thus, the hash scheme selects $d = 6$ backyard buckets and maps the cyan virtual page to a frame in the least full backyard.

theoretical guarantees hold with high probability over the random choices made by the algorithm, i.e., the choice of hash function, for any set of page requests made without knowledge of the algorithm's randomness (i.e., the hash function). We measure empirically in Section 4.2 that δ is roughly 2%. Moreover, the hash table is stable and has associativity $h = f + d \cdot O(\log\log p)$. Thus Iceberg hashing is an ideal candidate to serve as the theoretical starting point for mosaic.

In our prototypes, we use front yard bins of size $f = 56$, backyard bins of size 8, and $d = 6$. As a consequence, the total associativity h is 104, so the number of bits in a CPFN is 7.

Mosaic frame allocation. Mosaic structures memory as an Iceberg hash table and uses Iceberg hashing for page allocation. Allocating a mosaic page is analogous to inserting into an Iceberg hash table: we hash the virtual address to a front yard bucket and, if there is room in the bucket, store the page there. Otherwise we attempt to store it in a backyard bucket chosen using the power of d choices.

2.4 Mosaic Swapping

When no slot is available for a new allocation, we must choose a page to evict. Specifically, we must choose a page from among the buckets that can be used for the new allocation. To solve this problem, we propose a new eviction algorithm, ***Horizon LRU***, that is designed specifically to work with Mosaic. Horizon LRU is based on an algorithm with strong bounds on its paging costs (relative to baseline LRU) and, as we will see in §4, good performance in practice. The naive solution of simply evicting the least-recently-used page in the target buckets does not have the same performance guarantees.

Horizon LRU builds on prior work [7], which described an efficient page-eviction algorithm for a low-associativity cache. The key

idea in prior work is to simply implement cache replacement as if the cache were slightly smaller (less than $1 - \delta$ in size), so that one never sees associativity conflicts, only capacity evictions. The total number of evictions will be the same as LRU running on a fully associative cache of size $(1 - \delta)p$. The downside of this algorithm is that it completely wastes a fraction δ of memory.

Horizon LRU extends the above ideas, making two improvements while preserving the theoretically backed structure of the page eviction algorithm.

First, Horizon LRU does not evict any page until absolutely necessary. Rather, it marks pages that would be evicted unnecessarily as ***ghost pages***. Ghost pages are kept in memory in case they are referenced again, but the page allocation algorithm treats them as if their frames are free. So, for example, if there is a ghost page in the front yard bucket for an allocation, then it actually evicts that page and uses its slot for the new allocation. In the backyard, ghost pages do not count towards a bucket's occupancy when choosing the least-occupied bucket in the power-of-d-choices algorithm.

Second, Horizon LRU exploits the local structure of the mosaic page-allocation scheme to implement LRU without having to maintain a global LRU list. Horizon LRU tracks the last time that each page was accessed, as well as a global time stamp, called the ***horizon***, which is the high-water mark of the access times of all pages it has evicted. All pages whose most recent access time is prior to the current horizon are ghosts.

Horizon LRU simulates a global LRU algorithm because every time we update the horizon, we are marking all the pages with access times older than the new horizon as effectively evicted—exactly the set of pages that a global LRU algorithm would have actually evicted.

2.5 Limitations and Extensions

There are some limitations that arise as a result of the dependence of PFNs on their corresponding VPNs and the process's ASID. In particular, the set of possible PFNs for one (ASID, VPN)-pair are generally completely distinct from those of a different (ASID, VPN)-pair. As a result, mosaic pages as described above do not support duplicate mappings of the same physical memory within a single address space, or page sharing across address spaces. Similarly, as described above, mosaic does not support multiple mappings of the same file, even within a single address space.

We see several options in a large design space of possible solutions to the page sharing problem, either through special casing or a layer of indirection. We present a few representative solutions below. Ultimately, determining the best course will require careful empirical evaluation to ensure a careful balance among competing concerns, including latency, space usage, application requirements, and chip area.

A simple way to support page sharing is to introduce special-casing for shared pages so that they use traditional (non-Mosaic) TLB entries.

There are also approaches that more directly integrate shared pages into the Mosaic framework. For example, rather than using (ASID, VPN) as the input to the hash function, we give each ToC a unique identifier, which we call its ***location ID***. Then, to determine the PFN of the ith page within any mosaic page, we hash (location ID, i) instead of (ASID, VPN). Now we can use the same ToC multiple times within a single process's address space (for, e.g. duplicate mmaps), or use it in different address spaces to create shared memory.

This approach has two costs. First, each TLB entry needs to store its ToC's location ID, so TLB entries get larger. Second, during address translation, the hash function cannot be evaluated until after the TLB lookup completes, whereas hashing (ASID, VPN) could be done in parallel with the TLB lookup. This could potentially increase the latency of address translations. We can compensate for the lack of parallelism as follows. First, we have the OS generate location IDs randomly. (Although this may cause a few ToCs to be assigned the same location ID, Iceberg hashing is robust enough to handle this.) Now we can use extremely simple, low-latency hash functions, because the inputs to the hash functions are already randomized.

One could also adapt techniques used in other contexts to address the problem. For instance, rather than assigning a unique location ID to every ToC, one could assign location IDs to each segment and use a modest table of segments to translate (ASID, VPN) pairs to (location ID, offset) pairs. Or one could adapt techniques for handling aliases in virtually indexed caches, which map one virtual address to another within the cache [64]. We leave it as future work to evaluate these approaches.

3 IMPLEMENTATION OF MOSAIC PAGES

This section describes the gem5-based mosaic hardware simulator and prototype mosaic page allocator in Linux that we implemented in order to evaluate the performance and feasibility of mosaic pages. Mosaic pages require changes to four system components: the TLB, the page table, the OS page allocator, and the OS page eviction algorithm.

3.1 Gem5 Full-System Simulation

We implement mosaic pages using gem5 [12], a widely used architectural simulator. Our implementation involves systematically redesigning both the TLB and parts of the traditional radix-tree page table, and provides a full system simulation with support for mosaic pages.

Mosaic TLB. We extend gem5 to support mosaic TLB entries, using front yard buckets of size $f = 56$, backyard buckets of size $b = 8$, and $d = 6$ choices of backyards. Thus the total associativity of the page allocation scheme is $56 + 8 \times 6 = 104$. We encode CPFNs into 7 bits as follows. An unmapped page is the all-ones CPFN. Otherwise, the leading bit indicates whether the page is mapped to the front yard or the backyard. If it is mapped to the front yard, then the remaining 6 bits indicate the frame offset within the bucket. If it is mapped to the backyard, then the next 3 bits indicate which of the 6 backyard buckets was chosen, and the last 3 bits indicate the frame offset within that bucket. We store a valid bit per sub-page, and we use additional bits for permissions and other metadata, at mosaic page granularity.

Our mosaic TLB model takes as a parameter the mosaic arity, which may lead to very wide TLB entries when configured with large arity. By default our simulator uses an arity of 4 and 7-bit

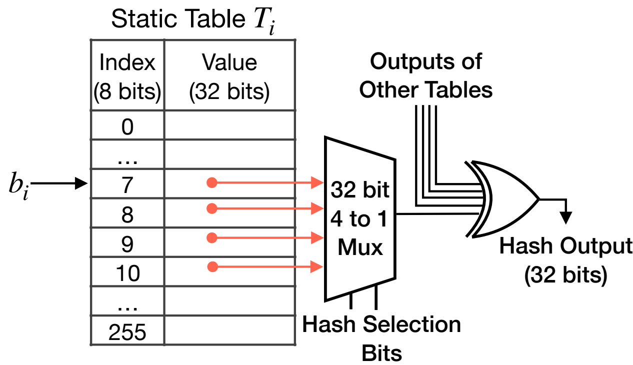


Figure 4: Hardware diagram for Tabulation Hashing with probing to produce four hash function outputs. Here the diagram for a single table is shown. We use one table for each byte of the VPN.

CPFNs. This yields ToCs of 28 bits, which mean that TLB entries are smaller than the 36-bit PFNs stored in most current x86 TLBs.

Note that the mosaic TLB design changes the update and decode logic for TLB entries (and maybe their width), but the mosaic memory mapping restrictions are orthogonal to the associativity of the TLB itself. Therefore a mosaic TLB can store the TLB entries using any caching design that it could use for a conventional TLB. So, for example, the TLB hardware could store TLB entries in a fully associative cache, a direct-mapped cache, or an N-level associative cache. We analyze the effect of different TLB associativity levels in Section 4.1.

When the OS invalidates a mapping of a sub-page within a mosaic page and invalidates the TLB entry, our TLB model only invalidates the sub-page's entry within the larger mosaic page's ToC. We do not invalidate the entire mosaic page's entry in the TLB. Upon a capacity miss, the TLB manages its own space using LRU to evict TLB entries for an entire mosaic page.

In our gem5 model, for ease of simulation, we maintain one TLB for the conventional (vanilla) mode and another TLB for the mosaic mode; results are computed for both modes simultaneously. Each memory access is fed to both TLBs with a separate page table walker for each TLB to handle misses. In mosaic mode, a conventional mapping consumes an entire TLB entry. We did this to keep the total number of TLB entries consistent for the evaluation, while compensating for the lack of sharing (i.e., shared pages still take up space in the TLB).

We further note that our model treats all shared pages as if they were copied and unshared in a mosaic address space, and are therefore compressible. None of our workloads use shared memory other than for library code, but it does mean that our results reflect a marginal trade of DRAM space for higher TLB reach.

Hash functions. In order to maintain TLB hit performance, we require a hash function that can run within the latency of the L1 TLB. Tabulation hashing is an established hashing technique that can produce high-quality hash values using small static tables [43]. Specifically, for an input I each byte $b \in I$ is an index into separate

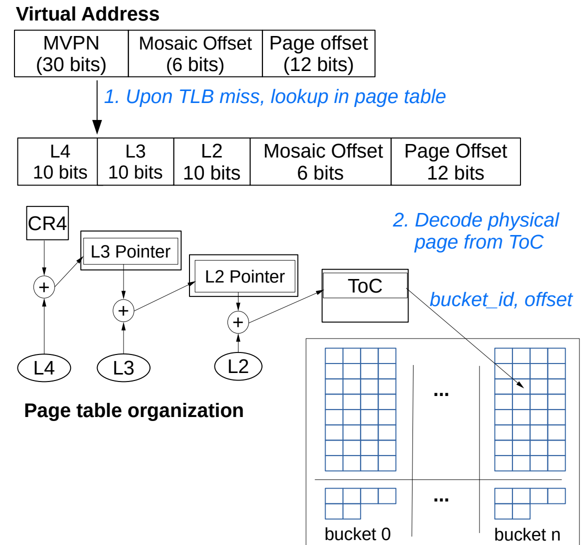


Figure 5: Upon a TLB miss, Mosaic walks the page table, which may be any structure; here, we present a traditional radix tree. As an optimization, the leaf nodes are modified to store tables of contents (ToCs), which are decoded to identify the physical location of the pages, using the same logic as TLB entries.

tables, each with 256 entries. The values returned by all these tables are then XOR'd together to give the output of the hash.

To produce multiple hash results from a single set of tables we probe from the value of I_b using the hash function id as an offset. For example, with 4 hash functions and a static table T, the hashes of an input I would be $H_0 = \bigoplus T_b[I_b]$, $H_1 = \bigoplus T_b[I_b + 1]$, $H_2 = \bigoplus T_b[I_b + 2]$, and $H_3 = \bigoplus T_b[I_b + 3]$. Probing in this way allows us to produce sufficiently random hash results without the need for additional tables (and thus power and chip area) for each hash function. This design is outlined in Figure 4.

Note that a hardware implementation can compute all the hash functions in parallel with the TLB lookup to obtain the CPFN. It can then use the CPFN to select the correct hash function's output and compute the PFN.

Mosaic page table structure and hardware. In large part, the changes required by mosaic pages are orthogonal to the page table design. In our simulation, we modify the leaves of the page table to map MVPNs to ToCs, illustrated in Figure 5, but we keep the overall radix-tree structure of the default gem5 page table. This structure allows the system to efficiently fetch and map a virtual mosaic page to ToC translation.

Our current prototype stores permission, present, accessed, and dirty bits in the page table for each encoded physical page in the ToC, even though permission bits must be identical within a mosaic page. The TLB currently only caches the present bit per CPFN, and other bits at the granularity of a mosaic page. We retain this flexibility in the page table to study the impact of this restriction in future work.

3.2 Mosaic Page Management in Linux

This section describes our prototype implementation of mosaic page management in Linux. This prototype is designed to run on unmodified hardware, so it implements only the restricted allocation and swapping mechanisms used in mosaic pages and uses the standard TLB and page table from x86 (in contrast to Section 3.1). However, all allocations and swaps obey the mosaic rules in Section 2, and so the prototype faithfully captures the conflicts that a full mosaic paging system would incur.

Specifically, we implement the mosaic page allocator in Linux kernel version 5.11.6. This allocator replaces the Linux allocator only for anonymous memory allocation. To support this, memory is split into two parts: at boot time a fixed amount (4 GiB) of physical memory is reserved for the mosaic page allocator, and the rest is used for the Linux default page allocator. We allocate anonymous, unshared memory for the process under test from this pool; all shared mappings, file caches, and the kernel are still placed in pages used by the standard Linux page allocator. We evaluate the system with applications whose space usage is largely anonymous, unshared mappings. Our prototype does not support inheriting mosaic pages via `fork()`, which would cause anonymous pages to become shared; our test applications do not use `fork()` to create child processes.

Our prototype Linux mosaic page allocator uses the same parameters as our gem5 simulator, i.e., front yard buckets of size $f = 56$, backyard buckets of size 8, and $d = 6$ choices. Each bucket contains a linked list of free pages. When a process requests an anonymous page, the allocator hashes the (ASID, VPN)-pair in order to obtain the buckets which can be used for allocation, as in Section 2. We use xxHash [13], a fast hash algorithm available in the mainline Linux kernel. The allocation happens when the process accesses the page because Linux uses demand paging.

Horizon LRU in Linux. We implement swapping using the Horizon LRU algorithm as described in Section 2.4.

One challenge is that Horizon LRU requires access timestamps, but current hardware maintains only access bits, not timestamps. A real mosaic page system would store timestamps instead of access bits, so this is only a challenge for our prototype, not of a real mosaic system. Our implementation uses the access bits to emulate up-to-date timestamps. We create a background daemon, which scans mosaic memory at regular intervals (1 s by default). If a page has been accessed since the last scan, we update the timestamp and clear the accessed bit. In the x86 architecture, a processor invalidates TLB entry once software clears the access bit in the corresponding page table entry. This leads to a high TLB misses and becomes an overhead. To alleviate the overhead, we use a sampling based approach [32]. For each page, we maintain 8 recent histories of access status, and classify the page is hot or cold. During scanning, we always read and clear the access bit of cold pages. For hot pages, we read and clear the access bit for 20% of pages and we consider remaining 80% of pages as accessed.

4 EVALUATION

This section presents experiments to answer the following questions about Mosaic performance:

Table 1: Experimental platform details.

(a) Gem5 Experiment Environment (§4.1)

Processors	Single core `TimingSimpleCPU`
Address sizes	36-bit VPNs and 36-bit PFNs
L1 DTLB	Unified TLB for 4 KiB and 2 MiB pages, 1024 entries, 1 to 1024-way (varied) set associative
L1 ITLB	Unified TLB for 4 KiB and 2 MiB pages, 1024 entries, 1 to 1024-way (varied) set associative
L1d cache	64 KiB, 2-way set associative
L1i cache	32 KiB, 2-way set associative
L2 cache	2 MiB, 8-way set associative
L3 cache	16 MiB, 16-way set associative
Memory	16GB DDR4
OS	Linux 4.16, Debian

(b) Linux Experiment Environment (§4.2–4.3)

Processor	Intel Xeon E3-1220 v6 (Sandy Bridge), 4 cores, 3.00GHz
Memory	32 GiB (4 GiB reserved for Mosaic)
OS	Linux 5.11.6, Ubuntu 20.04
Swap Device	4 GiB Ramdisk

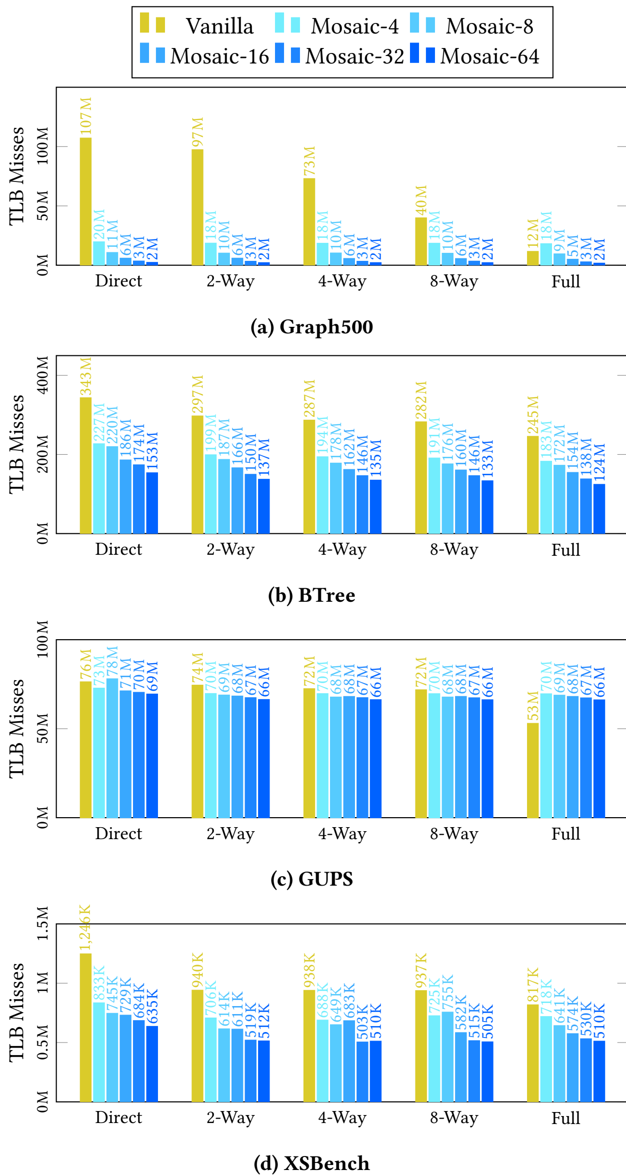
Table 2: Workloads used for evaluating hardware TLB and OS designs.

Workload	Description	Memory Footprint	Instructions (billions)
Graph500	Parallel graph processing benchmark.	1010 MiB	≈ 16.0
BTree	Benchmark for index lookups on a B+ Tree data structure.	2618 MiB	≈ 54.3
GUPS	Microbenchmark that generates random accesses, resulting in high TLB misses.	8207 MiB	≈ 23.2
XSBench	HPC benchmark that represents a key computational kernel of the Monte Carlo neutron transport algorithm.	1012 MiB	≈ 20.0

- Does Mosaic reduce TLB misses? (§4.1)
- Does Mosaic reduce memory utilization? (§4.2)
- Does Mosaic increase swapping? (§4.3)
- Is Mosaic hardware feasible? (§4.4)

4.1 Does Mosaic Reduce TLB Misses?

Table 1a describes the system that we simulated using gem5. We vary the TLB in two dimensions. First, we vary the mosaic arity from 4 to 64, *i.e.*, we vary the size of mosaic pages from 16 KiB to 256 KiB. Second we vary the associativity of the TLB from direct-mapped to fully associative.



(a) Graph500

(b) BTree

(c) GUPS

(d) XSBench

Figure 6: TLB Misses on the Graph500, BTree XSBench, and GUPS workloads with Mosaic and Vanilla TLBs with different-sized tables of contents (ToC) and set-associativity.

To study the TLB performance of Mosaic compared to a standard "vanilla" TLB, we run four widely used workloads, Graph500, BTree, GUPS, and XSBench using full system gem5 hardware simulation. Table 2 shows the total instructions simulated.

Figure 6 shows the number of TLB misses incurred during each workload. Vanilla represents standard Linux and x86 page tables. Although we disabled huge pages for application use, the Linux kernel itself was mapped using huge pages in vanilla, giving vanilla a slight advantage over mosaic.

The high-level take-away is that mosaic pages can reduce TLB misses across a wide variety of workloads and TLB associativities. In many cases, Mosaic can reduce TLB misses by a dramatic amount, *e.g.*, almost completely eliminating them in Graph500 and XSBench, reducing them by up to about half in B-Tree and about a quarter in GUPS.

Mosaic arity. In this experiment, we measure the sensitivity of the TLB misses to the number of CPFNs that are in one TLB entry (arity). We vary the number of CPFNs from 4 to 64. Note that with an arity of 4, all 4 CPFNs fit in a single unmodified x86 TLB entry. Even with an arity of only 4 (Mosaic-4), Mosaic shows a substantial reduction of 6–81% in TLB misses for Graph500, BTree, and XSBench workloads, and with an arity of 64 (Mosaic-64) reduces misses by 11–98%. Mosaic shows less improvement on GUPS, which is unsurprising, because GUPS is a synthetic benchmark designed to stress the system with extremely random memory accesses.

TLB associativity. As Figure 6 shows, increasing TLB associativity reduces TLB misses for all approaches due to reduced conflict misses. However, switching from a vanilla TLB to a mosaic TLB is far more effective at reducing TLB misses than increasing the associativity of the TLB. For example, observe that a direct-mapped Mosaic-8 TLB outperforms a fully associative vanilla TLB for Graph500, BTree, and XSBench. In fact, the performance of Mosaic is not significantly impacted by TLB-associativity indicating that a Mosaic system could use more efficient lower-associativity TLB designs.

We note that vanilla with a fully associative TLB out-performs Mosaic-4 on Graph500 and Mosaic-4–16 on GUPS. This is because vanilla Linux uses huge pages to map the kernel, effectively gaining a slightly larger TLB than Mosaic. Smaller experiments with huge pages fully disabled remove this artifact (giving all Mosaic arities fewer TLB misses than vanilla), but we were not able to complete all simulations with this change in time for the final paper deadline.

4.2 Does Mosaic Reduce Memory Utilization?

In this section we empirically measure δ, the memory overhead discussed in Section 2, and compare this to the memory utilization achieved by the default Linux virtual-memory subsystem.

The hardware and software used in these experiments are presented in Table 1b. Each data point reports the average and standard deviation of ten runs, unless otherwise stated. We use the `seq-csr` implementation of Graph500 unless otherwise stated. To reduce noise from various processes in the system, we use the Mosaic page allocator only for the benchmark process. All other processes on the system use the default Linux allocator.

We scale the workloads down to approximately 4 GiB and limit memory for the test workload to 4GiB, in order to work around missing features in our current prototype, such as the inability to put the kernel or shared mappings in mosaic pages. The other 28 GiB are available only for system background processes, not the benchmark. We limit application memory on unmodified Linux using cgroups. We restrict the amount of mosaic memory in our prototype to 4 GiB, except for a small amount of file memory (~400 KiB), which we could not easily restrict in the mosaic case. This memory split is an experimental artifact, not a design feature.

As discussed in Section 2, Mosaic may start to have associativity conflicts once it hits a memory utilization of $1-\delta$. However, memory

Table 3: Memory utilization under the Mosaic page allocation at the point of the first associativity conflict, and the steady-state utilization over the entire workload. The standard Linux allocator begins swapping at about 99.2% memory utilization, so Mosaic has essetially no memory overhead in these benchmarks.

Workload	Footprint	First associativity conflict $(1-\delta)$	Steady-state utilization
Graph500	4158 MiB	98.07% ±0.04	99.22% ±0.01
XSBench	4156 MiB	98.02% ±0.08	99.21% ±0.01
BTree	4154 MiB	98.03% ±0.10	99.21% ±0.01
Graph500	4413 MiB	98.02% ±0.06	99.79% ±0.01
XSBench	4412 MiB	98.04% ±0.13	99.73% ±0.01
BTree	4409 MiB	98.00% ±0.06	99.74% ±0.01
Graph500	4669 MiB	98.05% ±0.04	99.99% ±0.00
XSBench	4668 MiB	98.01% ±0.02	99.92% ±0.01
BTree	4664 MiB	98.04% ±0.04	99.96% ±0.01
Graph500	4924 MiB	98.05% ±0.03	100.00% ±0.00
XSBench	4924 MiB	98.05% ±0.07	99.98% ±0.01
BTree	4919 MiB	98.00% ±0.07	99.99% ±0.00

utilization can go beyond $1-\delta$ due to ghost pages (§2.4). To evaluate these two effects, we measure both the memory utilization when our benchmark experiences its first associativity conflict and its steady-state memory utilization over the entire benchmark run.

We measure this by running Graph500, XSBench, and BTree configured to have memory footprints between 4.06GB to 4.80GB, so that, in a fully associative paging scheme, memory would be nearly 100% utilized. Table 3 presents the memory utilization at the time of the first associativity conflict $(1-\delta)$ and the steady-state memory utilization.

As the table shows, the first conflict appears at around 98.03% utilization across all workloads, indicating that δ is roughly 2%. In contrast, we observed that Linux with the default allocator began swapping once memory utilization reached 99.2% (because each memory zone has its own watermark). Thus Mosaic's associativity restrictions do not cause Mosaic to begin swapping significantly sooner than with the default Linux allocator. Furthermore, over the entire execution, the workloads are able to utilize over 99.2% of available memory, and this increases as the footprint increases, so the overall memory overhead of Mosaic is less than 1%.

4.3 Does Mosaic Increase Swapping?

We run each workload with a variety of memory footprints, from just over the size of available memory to about 57% larger, and report the total number of swap I/Os as reported by `sysstat`. We use the same hardware and software as in the previous section. We present the average of 5 runs. Standard deviations were all below 5%.

Table 4 compares the behavior of Mosaic's swapping algorithm, Horizon LRU (§2.4), to the default Linux implementation. The difference column gives the percent reduction in swapping I/O performed by Mosaic (*i.e.* higher is better). Green cells indicate that Mosaic performed less swapping, and red indicates that it performed more. When the workload is just slightly over the size of available memory, Mosaic performs more swapping than the default Linux allocator. This is because, as we showed in Section 4.2, Linux is able to utilize about 1% more memory than Mosaic. However, once the workload footprint passes this edge case, Mosaic matches or outperforms Linux, sometimes by a significant amount (up to 29% in the best case). This may be because the associativity restrictions in Mosaic slightly perturb LRU's choices, preventing it from suffering from well-known bad cases, such as cyclic memory references.

Table 4: Number of memory swapping operations while increasing the workloads sizes

Workload	Memory Footprint (MiB)	Linux 4096 MiB (K pages)	Mosaic 4096 MiB (K pages)	Difference (%)
Graph500	4158	1031.10	2043.86	-98.22
	4413	29908.91	24644.36	17.60
	4669	62785.90	44585.39	28.99
	4924	79450.93	57818.43	27.23
	5180	82670.25	68392.23	17.27
	5436	85026.79	79061.77	7.02
	5691	93284.21	90125.03	3.39
	5947	104071.54	102121.28	1.87
	6203	114875.52	113780.59	0.95
	6459	125003.06	119083.55	4.74
XSBench	4156	337.55	391.79	-16.07
	4412	1465.91	1279.11	12.74
	4668	2726.31	2404.34	11.81
	4924	4071.57	3693.00	9.30
	5180	5416.14	5043.78	6.88
	5435	6778.77	6443.30	4.95
	5691	8141.77	7910.56	2.84
	5947	9538.62	9421.29	1.23
	6203	10802.44	10678.59	1.15
	6459	11871.56	11888.83	-0.15
BTree	4154	983.57	1166.02	-18.55
	4409	5790.18	5127.18	11.45
	4664	11609.53	10338.29	10.95
	4919	18142.24	15605.34	13.98
	5175	25256.59	21081.30	16.53
	5430	32684.41	26973.25	17.47
	5685	40369.66	33539.42	16.92
	5940	48305.64	41217.21	14.67
	6196	55108.28	48766.51	11.51
	6451	60877.65	54952.76	9.73

4.4 Hardware Evaluation

We used a two-step hardware evaluation to assess the viability of our approach. Our first set of evaluations were performed using an FPGA fabric. We then performed an evaluation using a 28nm CMOS process/foundry with more detailed timing analysis.

For our FPGA implementation, we evaluated the latency and area impact of adding hashing to the TLB hit path, as described in

Table 5: Size and latency of Tabulation Hash circuit on an FPGA given a number of hash functions.

H	LUTs	Registers	F7 Mux	F8 Mux	Latency
1	858	32	0	0	2.155ns
2	1696	32	32	0	2.155ns
4	3392	32	64	32	2.155ns
8	6208	32	2880	160	2.155ns

§3.1 in Verilog. Synthesizing for an Artix-7 FPGA, we used simple lookup tables to implement the static tables. We found the latency of the Tabulation Hash circuit to be 2.155ns or a clock frequency of 464MHz. When varying the number of hash functions from 4-8, the clock frequency of the circuit was unchanged. Increasing the number of hash functions increases the number of values extracted from the static tables and the size of the post-table muxes. As the latency of the circuit is unchanged, generating multiple hash outputs by probing as described in §3.1 is an efficient approach. Furthermore, the Tabulation hash circuit with eight hash functions requires 6208 Slice LUTs, 32 Slice Registers, 2880 F7 Muxes, and 160 F8 Muxes. Table 5 gives the sizes of the circuit with other quantities of hash functions.

Encouraged by these results, we also then implemented our hardware changes in System Verilog and synthesized it using a commercial 28nm CMOS process. We implemented the static tables as registers, and used Cadence synthesis tools with standard cell libraries to generate results. We present results for the worst-case variation corner (i.e., Tr_{FF}, Vdd_{MIN}, and RC_{BEST}, at 1V Vdd, 125 °C).

The synthesized circuit ran at a maximum frequency of 4 GHz and a latency of 220 ps and 20 picoseconds positive slack. Additionally, increasing the number of hash functions did not increase the latency while increasing the area minimally. This increase in area is due to larger muxes needed to select from more values looked up from the table. Therefore, generating multiple hash outputs by probing as described in §3.1 is an efficient approach. The design uses 13.806 KGE area for 8 hash functions, normalized to a 2-input NAND gate in our process node when using 8 hash functions.

5 RELATED WORK

Because virtual memory is a nearly ubiquitous abstraction for programmers, ensuring low address translation overheads is a perennial goal. Prior research on navigating the tension between increasing TLB reach and the costs of defragmentation spans decades. This section describes related work on improving TLB reach through larger pages and more efficient defragmentation, as well as MMU caching. Like Mosaic, hashed page tables use hashing to accelerate address translation, although hashed page tables are intended to reduce TLB miss costs and total space overhead of page tables, whereas mosaic increases TLB reach. We also summarize prior research that reduces the associativity of memory mapping.

5.1 Huge Pages and Defragmentation

On current hardware, one must balance the CPU costs and memory bandwidth of defragmentation against potential gains from huge pages, as one can easily squander the performance gains from huge pages on defragmentation overhead [32, 39, 66]. Several works have proposed techniques that balance the costs of defragmenting memory in the OS with gains from TLB coverage [32, 37, 62, 66]. A related issue is ***memory bloat***—when an application maps huge pages but would use considerably less space with smaller pages. Memory bloating increases memory pressure and can induce swapping [37]; for this reason, many databases recommend disabling huge pages [36, 48, 51, 60]. An underlying reason for both issues is that performance gains from contiguity are all-or-nothing.

In order to mitigate defragmentation costs and memory bloat, one can tolerate some discontinuity in the physical memory backing a huge page, such as by adding an intermediate address translation layer and augment hardware [16, 40, 53, 65]. A recent example is perforated pages [40], which start with a 2 MiB huge page but allow the OS to redirect individual 4 KiB "holes" to different physical pages. This requires an extended page table structure to map these hole pages and extend both the L1 and L2 TLB with bitmaps to filter holes and trigger additional page table traversals. This indirection is also sufficient to handle issues such as sharing regions of memory across processes or unusable regions of DRAM ("dead pages"). The performance intuition behind these techniques is that if one could not form an entire huge page, but had enough contiguity to split the virtual huge page into a small number of discrete physical extents, this is an improvement over falling back to strictly using 4 KiB pages. Mosaic also composes larger pages from dis-contiguous physical pages; a key distinction is that mosaic leverages a low-associative hashing scheme to reduce hardware costs while providing sufficient flexibility for software.

In summary, the road to huge pages has not been smooth. Despite the appeal and apparent simplicity of huge pages and decades of research, transparent support for 2MiB pages is still under active development, and modern OSes do not transparently support 1 GiB pages yet.

5.2 Redesigning TLB Layout

Alternatively, there are approaches to increase the TLB reach by redesigning the TLB layout. A related strategy to leverage smaller contiguity than a huge page is to *coalesce* adjacent TLB entries into one entry if the entries happen to be both virtually and physically contiguous [45]. This approach adds modest hardware to infer when TLB entries can be coalesced; later work shows OS support for coalescing [42]. For TLB coalescing, the performance gains are proportional to the amount of physical and virtual contiguity, and unlike standard huge pages, this approach does not depend on 2MiB increments in contiguity.

In a cache design, subblocking (or sectoring) is a technique to leverage spatial locality to cache a larger block from a set of subblocks [23, 33]. Correspondingly, there are approaches to apply subblocking to TLB design [55]. Some MIPS processors have a TLB with a subblock factor of two, which means single TLB entry stores two different PFNs for two adjacent VPNs [22], at a cost of widening the TLB entry by the additional PFNs. Mosaic can be viewed as a TLB with a subblocking factor of *arity* (four in our prototype), combined with PFN compression, which ensures that Mosaic pages fit within the same TLB entry width.

Perhaps most similar to this work are several papers based in part on the observation that even non-contiguous pages are often

nearby in physical memory, enabling a form of prefix compression by sharing the upper bits in the physical address and packing the differing, lower bits into the entry [2, 44, 57]. All these papers and Mosaic improve TLB reach when virtual mappings are constrained. Although these papers relax the contiguity requirements, Mosaic removes the need for contiguity altogether.

Another approach to economize TLB entries is by introducing *segmentation*, specifically on dedicated servers without multi-tenancy and workloads size matched to server memory capacity. Basu et al. [5] show that a single, variable-sized segment can eliminate nearly all TLB misses for a set of common workloads by increasing virtual and physical contiguity. This idea has been extended with multiple segments (or ranges) [28, 41]. However, these approaches leverage deployment-specific characteristics, such as no swapping, that would not necessarily be true of all deployments.

Orthogonal designs have explored increasing TLB reach by moving the TLB off of the hit path for the L1 cache and onto the miss path. However, for this to work, caches accessed without the TLB must be indexed by a virtual address rather than a physical address, called ***virtual caching*** [6, 18, 30, 41, 61]. However, virtual address indexing complicates shared data handling. In contrast, mosaic techniques increase the reach of a TLB, regardless of whether the TLB is on the L1 cache hit path; however, the need to compress physical addresses is more acute when the TLB is accessed on the L1 hit path.

5.3 Limited Associativity VM

Reducing the associativity of virtual memory mappings has been used to optimize CPU cache behavior. Page coloring [31, 58] is a technique to ensure that sequential virtual pages will not contend for the same cache line. However, the introduction of multi-way set-associative caches reduces the demand for page coloring. As a result, some OSes, including Linux, do not support page coloring [24]. Although page coloring restricts the physical page placement, the resulting associativity is much higher than Mosaic, *e.g.,* in a system with 4 GiB RAM with 128 colors, each virtual page can be mapped to 8,192 physical page frames. Because Mosaic's mapping restrictions are more stringent than page coloring, it is unlikely that the two techniques can be profitably combined. However, Mosaic's randomization of virtual-to-physical mappings may be sufficient in expectation to avoid the cache pathologies prevented by page coloring, which we leave for future work.

Several works studied how to use main memory as set associative caches. Alan Jay Smith studied the performance of set associative main memory mapping, finding a small increase in the number of page faults compared to LRU in a fully associative memory mapping design [50]. More recently, Picorel et al. [47] proposed set-associative memory for near-memory processors. They observed that a 4-way set associative memory can eliminate most conflict misses of a single process application. Conventionally, restricting physical page placement comes with a cost of a memory underutilization. To avoid this issue, Utopia [27] splits system memory into multiple segments, flexible segments and restrictive segments; a restrictive segment is organized as a set associative memory. Utopia measures the number of TLB misses for each page and records the counter in the page table entry. Pages with a high TLB miss count are migrated into a restrictive segment. In summary, prior work has shown that the flexibility of fully associative memory comes at a cost, and that this high degree of flexibility may not be strictly necessary.

5.4 MMU Caching

A complementary approach to increasing TLB reach is reducing the costs of a TLB miss. Specifically, by caching portions of the page tables in hardware ***MMU caches*** [3, 10, 11], one can potentially eliminate a series of sequential loads to walk the page table data structure. These caches are also amenable to common optimizations such as pre-fetching and speculation [4, 46], which can further reduce the TLB miss penalties. The effectiveness of these caches is also a function of the complexity of the underlying mapping structure: these caches are very effective for nested hardware page tables [1, 4, 10], where each memory access in the guest triggers a page table walk in the supervisor, Translation-triggered prefetching observes that by integrating MMU caches with data caches, one can use TLB miss information as a hint to prefetch the related *data* into the data caches, with very high accuracy. This prefetching can be further improved with modest changes to how the OS places page tables in physical memory [34]. The primary downside of these techniques is that they all require more chip area and power.

5.5 Hashed Page Tables

Hashing has also been used in page tables [15, 17, 21, 25, 26, 56]. In a hashed page table, upon a TLB miss, the CPU hashes the requested virtual address and process identifier to identify a bucket of page table entries. Ideally, this bucket holds only the target page table entry, and requires only one memory reference to determine the physical address. Much more commonly, this bucket stores a collision chain of such page table entries, requiring several memory references to identify the desired page table entry [3]—easily eroding the best-case gains in the worst cases. Recent work [49, 52, 63], however, suggests that these shortcomings may not be fundamental to the hashed page table approach, but rather to the choice of hashing schemes. More recent work also points out the opportunity for nested, hashed page tables to unlock more parallelism in the TLB miss path than a radix-tree page table structure [52].

A key distinction between Mosaic and hashed page tables is that Mosaic uses hashing to increase TLB reach (and, thus, the TLB hit rate), whereas hashed page tables reduce the TLB miss cost. Mosaic is compatible with any page table design, and in our prototype, we retain radix-tree page tables.

Mosaic's iceberg hashing may improve upon cuckoo hashing in the page table, which we will explore in future work. However, we do note that in cuckoo hashing-based page tables, the tables must be resized after they reach an occupancy threshold (60% occupied in Elastic Cuckoo Tables); at this point, page contents must be copied to new frames. A key improvement of Iceberg hashing is that it is stable and has a high load factor: even when memory is nearly full, page contents do not need to be copied to new frames to ensure the same performance as when the memory is not full. A key contribution of the mosaic design is delaying conflicts (and conflict resolution) until the system would swap anyway.

6 CONCLUSIONS

This paper shows how one can compress physical addresses in the TLB, thereby reducing TLB misses for big data workloads by 6–81% with comparable hardware, and even further with wider TLB entries. Many techniques for increasing TLB reach rely on physical contiguity, whereas Mosaic does not require contiguity or defragmentation. Moreover, we show that these constrained mappings do not induce additional swapping on average. Key to these results is a hashing scheme with the right properties for address translation: a high load factor, stability, and relatively few choices. Finally, Mosaic is compatible with many of the techniques in the literature to increase TLB reach, and these techniques can be profitably composed in future work.

DATA-AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Zenodo at http://doi.org/10.5281/zenodo.7709303, reference number [67].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on prior drafts of the work. We thank Montek Singh for assistance with the Verilog/FPGA toolchain. We thank Rajit Manohar for giving us access to the physical synthesis flow used for our hardware evaluations. Part of this work was completed while Mubarek and Mukherjee were at UNC. This work was supported in part by NSF grants CNS-1700512, CCF-1716252, CCF-1725543, CSR-1763680, CNS-1910593, CCF-1916817, CNS-1938709, CSR-1938180, CNS-1938709, CCF-2106827, CCF-2106999, CCF-2118620, CCF-2118830, CCF-2118832, CCF-2118851, CCF-2119300, CNS-2154771, 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 the United States Air Force Artificial Intelligence Accelerator 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.

A ARTIFACT APPENDIX

A.1 Abstract

There are three artifacts for this paper: a Gem5 model to reproduce Figure 4, a modified Linux kernel to reproduce Tables 3 and 4, and Verilog code to reproduce Table 5. The Linux artifact includes scripts to setup a KVM environment with Mosaic and vanilla Linux kernels. The artifact also includes scripts to run the Linux workloads in a VM and a script to generate tables.

A.2 Artifact Check-List (Meta-Information)

Gem5.

- **Program: Linux**
- **Compilation: The artifact includes a script to compile the modified gem5 simulator, create a QEMU-based virtual machine, and compile the Linux kernel.**
- **Binary: Binaries for Graph500, XSBench, BTree, and GUPS bench-marks are included.**
- **Run-time environment: Experiments are run in a virtual machine, on top of the gem5 simulator, which runs on bare metal.**
- **Hardware: A Linux system with 4 to 8 Intel CPUs, 20GB of free RAM, and superuser access permission to install packages.**
- **Output: The output generates regular Gem5 logs, summarized TLB statistics, and filtered TLB miss rate information for vanilla and Mosaic designs. The artifact describes the output files and information.**
- **Experiments: The experiments measure the total TLB accesses and total TLB misses incurred.**
- **How much disk space required (approximately)?: 20-30GB disk for sequential runs, and 100GB for parallel runs.**
- **How much time is needed to prepare workflow (approximately)?: 30 minutes**
- **How much time is needed to complete experiments (approximately)?: The artifacts have two types of input: (1) a tiny input to check if everything works, which can complete within 2 hours; (2) a large input used in the paper, which can take about 400 hours (a week if all steps are followed correctly) to complete for all data points.**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPLv2**
- **Archived (provide DOI)?: Yes https://doi.org/10.5281/zenodo.7592600**

Linux.

- **Program: Linux**
- **Compilation: The artifact includes a script to compile the modified Linux kernel.**
- **Binary: Binaries (and source) for Graph500, XSBench, and BTree benchmarks are included.**
- **Run-time environment: The experiments can be run either in a bare-metal system or in a VM.**
- **Hardware: A Linux system with more than 12GB of RAM and 2-cores of CPU is required.**
- **Execution: The artifact includes a set of scripts to execute the experiments.**
- **Output: The artifact will produce two tables in form of CSV documents.**
- **Experiments: The experiment will measure the number of swapping with provided benchmarks.**
- **How much disk space required (approximately)?: 15 GiB**
- **How much time is needed to prepare workflow (approximately)?: 30 Minutes**
- **How much time is needed to complete experiments (approximately)?: 20 Hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPLv2**
- **Archived (provide DOI)?: Yes https://doi.org/10.5281/zenodo.7592600**

Verilog.

- **Program: Verilog**
- **Compilation: Vivado**

- **Binary: All source Verilog, constraints, and memory files are included for the experiment**
- **Run-time environment: The experiment can be run either in a bare-metal system or VM**
- **Hardware: A Windows computer with at least 80 GB of free disk space, or an external storage device with at least 80 GB of free space**
- **Execution: The artifact includes a tutorial on Vivado installation and step by step instructions on how to run the experiment**
- **Output: Multiple reports will be generated by the experiment in Vivado. 2 of these reports will contain the experimental results we are concerned with**
- **Experiments: The experiment will measure latency and used resources of a tabulation hash function implementation**
- **How much disk space required (approximately)?: 80 GB**
- **How much time is needed to prepare workflow (approximately)?: 1-2 Hours for Vivado Installation**
- **How much time is needed to complete experiment (approximately)?: 1 Hour After Vivado Installation**
- **Publicly available?: Yes**
- **Archived (provide DOI)?: Yes https://doi.org/10.5281/zenodo.7592600**

A.3 Description

A.3.1 How to Access. The artifacts are available on GitHub: https://github.com/oscarlab/mosaic-asplos23-artifacts

A.3.2 Hardware Dependencies. For the Linux experiments, a system with an x86-64 CPU is required. The minimum system needs 2 cores of CPU and 12GB of RAM, and 15GB of available disk space.

The instructions for the gem5 and verilog experiments are tailored to and tested on an x86 host system, but may work on other ISAs.

To setup and run the Verilog experiment following the provided instructions a Windows computer with at least 80 GB of free disk space is required

A.3.3 Software Dependencies.

Gem5. We run workloads in a Ubuntu 18.04 or 20.04 system

Linux. We run workloads in a Ubuntu 20.04 system.

Verilog. Microsoft Windows and Vivado 2020.2 (Vivado HL WebPACK, Vivado Design Suite, Vivado, Vitis HLS, DocNav, Artix-7 devices, and cable drivers).

A.4 Installation

Gem5. The artifact includes scripts to compile the gem5 simulator along with automated steps to install packages, compile the Linux kernel required for running Gem5 VM, and scripts to prepare a VM image. All compilation scripts are clubbed into one (`compile.sh`), and the QEMU VM image preparation script can be found in `create_qemu_img.sh`. There are other helper scripts, as mentioned in the artifact README file.

Linux. The artifact includes scripts to prepare a VM image and build kernels. To create a VM image, use `create_kvm_disk.sh`. And to build mosaic and vanilla kernels, use `get_kernel.sh`.

Verilog. The README documents show how to download the Vivado tools from the Xilinx website and install them on a Windows system.

A.5 Experiment Workflow

Gem5. While there are several supporting steps (please see the README file), to launch a gem5 simulation using full system simulation the following command is used:

```
$ test-scripts/prun.sh $appname $associativity
  $tocsize $port $inputsize
```

Linux. To launch a KVM with a mosaic kernel:

```
$ ./run_qemu.sh -w
```

To launch a KVM with a vanilla kernel:

```
$ ./run_qemu.sh -wv
```

Verilog. The README describes how to synthesize the included verilog files, and run a timing and resource analysis, which generates multiple reports that contain the latency and resource usage results we presented

A.6 Evaluation and Expected Results

Gem5. The artifact includes one main script to launch a QEMU VM and run the experiments run a workload `test-scripts/prun.sh`. To reduce the execution time, the artifact feeds each memory reference to a vanilla TLB as well as Mosaic's TLB, and the TLB miss rate for vanilla and Mosaic are simultaneous displayed on the screen after execution.

Further, because each data point (associativity and TOC size combination) can take days for full system execution, the README describes how to perform parallel execution. Further, the detailed result files can be seen in the result folder, which uses the following structure to save the output:
`result/$app/mosaic/$associativity/$tocsize`
More details can be found in the artifact's README.

The expected output should be within 1–2% on shorter experiments, and within 10% on longer-running experiments.

Linux. The artifact includes two scripts to run workloads in a cgroup or in a Mosaic environment. Launch `run-cgroup.sh` or `run-mosaic.sh` script according to the running kernel. After getting results from both experiments, use `copy_kvm_homedir.sh` to copy files from the VM to the host. Then use `process.sh` to generate tables in csv form.

The amount of swapping varies from run to run; on our test systems, we saw a typical standard deviation less than 5%.

Verilog. The output of the analysis should be visible as a report in the verilog tools. Under "Route Design" you should find a "Timing Summary - Route Design" file. This file will should state the worst slack and total violation is -0.155ns. Since the design was synthesized using a clock with a period of 2ns, this means the minimum operational clock period of the Verilog design is $2ns + .155ns = 2.155ns$, which implies a maximum operational clock frequency of 464 MHz. Resource utlization can be found in the tabulationHash4_utilization_placed.rpt in reports.

We expect these results to be deterministic with respect to the input parameters.

A.7 Experiment Customization

Gem5. Our scripts support easy customization of simulation parameters including the table of contents (ToC) size (for Mosaic only), the associativity of the TLB, and the application to run. The scripts can be edited to adjust application parameters, or other architectural parameters we did not vary in the paper.

REFERENCES

[1] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2015. Fast Two-Level Address Translation for Virtualized Systems. *IEEE Trans. Comput.* 64, 12 (dec 2015), 3461–3474. https://doi.org/10.1109/tc.2015.2401022

[2] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*. IEEE, IEEE, Virtual Event, 515–528. https://doi.org/10.1109/ISCA45697.2020.00050

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[6] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 297–308.

[7] 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* (Virtual Event, USA) *(SPAA '21)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/3409964.3461814

[8] Michael A. Bender, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2021. All-Purpose Hashing. https://doi.org/10.48550/ARXIV.2109.04548

[9] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Tiny Pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '23)*. Society for Industrial and Applied Mathematics, USA, 477–508. https://doi.org/10.1137/1.9781611977554.ch21 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611977554.ch21

[10] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[11] Abhishek Bhattacharjee. 2013. Large-Reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[13] Yann Collet. 2016. xxHash: Extremely fast hash algorithm. https://cyan4973.github.io/xxHash/.

[14] Intel Coorporation. 2022. Intel 64 and IA-32 architectures optimization reference manual.

[15] Cort Dougan, Paul Mackerras, and Victor Yodaiken. 1999. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 229–237. https://doi.org/10.5555/296806.296833

[16] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. 2015. Supporting Superpages in Non-Contiguous Physical Memory. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. IEEE, USA, 223–234. https://doi.org/10.1109/hpca.2015.7056035

[17] Stephane Eranian and David Mosberger. 2000. The Linux/ia64 Project: Kernel Design and Status Update. *HP LABORATORIES TECHNICAL REPORT HPL* 85 (2000).

[18] James R. Goodman. 1987. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. ACM, ew York, NY, USA, 72–81. https://doi.org/10.1145/36206.36186

[19] Mel Gorman. 2010. Linux Huge Pages. https://lwn.net/Articles/375096/.

[20] Mel Gorman. 2018. AMD Zen Architecture. https://en.wikichip.org/wiki/amd/microarchitectures/zen.

[21] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. 2005. Itanium: A System Implementor's Tale. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 264–278.

[22] Joe Heinrich et al. 1994. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc.

[23] Mark D. Hill and Alan Jay Smith. 1984. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. ACM, New York, NY, USA, 158–166. https://doi.org/10.1145/800015.808178

[24] Michal Hocko and Tomas Kalibera. 2010. Reducing Performance Non-Determinism via Cache-Aware Page Allocation Strategies. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering* (San Jose, California, USA) *(WOSP/SIPEW '10)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/1712605.1712640

[25] Jerry Huck and Jim Hays. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/165123.165128

[26] Bruce L. Jacob and Trevor N. Mudge. 1998. A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/291069.291065

[27] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, Can Firtina, Rachata Ausavarungnirun, Nastaran Hajinazar, Jisung Park, Nandita Vijaykumar, and Onur Mutlu. 2022. Utopia: Efficient Address Translation using Hybrid Virtual-to-Physical Address Mapping. https://doi.org/10.48550/arXiv.2211.12205 arXiv:2211.12205

[28] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[29] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Energy-efficient address translation. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*. IEEE, USA, 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[30] Stefanos Kaxiras and Alberto Ros. 2013. A New Perspective for Efficient Virtual-Cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 535–546. https://doi.org/10.1145/2485922.2485968

[31] Richard E Kessler and Mark D Hill. 1992. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems* 10, 4 (nov 1992), 338–359. https://doi.org/10.1145/138873.138876

[32] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and*

Implementation (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, USA, 705–721. https://doi.org/10.5555/3026877.3026931

[33] John S. Liptay. 1968. Structural Aspects of the System/360 Model 85: II the Cache. *IBM Systems Journal* 7, 1 (mar 1968), 15–21. https://doi.org/10.1147/sj.71.0015

[34] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. ACM, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[35] Chris Mellor. 2022. SK hynix announces CXL 2 memory cards and SDK. https://blocksandfiles.com/2022/08/02/sk-hynix-announces-cxl-2-memory-cards-and-sdk/.

[36] MongoDB 2022. "Disable Transparent Huge Pages (THP)". https://www.mongodb.com/docs/manual/tutorial/transparent-huge-pages/.

[37] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, Massachusetts) *(OSDI '02)*. USENIX Association, USA, 89–104. https://doi.org/10.5555/1060289.1060299

[38] Prashant Pandey, Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance PMEM Hash Tables Through Stability and Low Associativity. In *Proceedings of the 2023 International Conference on Management of Data, to be published* (Seattle, WA, USA) *(SIGMOD '23)*. ACM, New York, NY, USA.

[39] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[40] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, Virtual Event, 913–925. https://doi.org/10.1109/ISCA45697.2020.00079

[41] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2016. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE, Seoul, Republic of Korea, 217–229. https://doi.org/10.1109/ISCA.2016.28

[42] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[43] Mihai Patrascu and Mikkel Thorup. 2011. The Power of Simple Tabulation Hashing. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (San Jose, California, USA) *(STOC '11)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1993636.1993638

[44] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA '14)*. IEEE, Los Alamitos, CA, USA, 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[45] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th International Symposium on Microarchitecture (MICRO-45)*. IEEE, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[46] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[47] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. 2017. Near-Memory Address Translation. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*. IEEE Computer Society, Los Alamitos, CA, USA, 303–317. https://doi.org/10.1109/PACT.2017.56

[48] Redis 2022. Redis Administration. https://redis.io/docs/manual/admin/.

[49] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. ACM, New York, NY, USA, 1093–1108. https://doi.org/10.1145/3373376.3378493

[50] Alan Jay Smith. 1978. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. *IEEE Transactions on Software Engineering* SE-4, 2 (mar 1978), 121–130. https://doi.org/10.1109/TSE.1978.231482

[51] Splunk 2021. Transparent huge memory pages and Splunk performance. https://docs.splunk.com/Documentation/Splunk/7.3.1/ReleaseNotes/SplunkandTHP.

[52] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2022. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. ACM, New York, NY, USA, 84–97. https://doi.org/10.1145/3503222.3507720

[53] Mark Swanson, Leigh Stoller, and John Carter. 1998. Increasing TLB Reach Using Superpages Backed by Shadow Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA '98)*. IEEE Computer Society, USA, 204–213. https://doi.org/10.1145/279361.279388

[54] Michael M. Swift. 2017. Towards $O(1)$ Memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. ACM, New York, NY, USA, 7–11. https://doi.org/10.1145/3102980.3102982

[55] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[56] M. Talluri, M. D. Hill, and Y. A. Khalidi. 1995. A New Page Table for 64-Bit Address Spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. ACM, New York, NY, USA, 184–200. https://doi.org/10.1145/224056.224071

[57] Xulong Tang, Ziyu Zhang, Weizheng Xu, Mahmut Taylan Kandemir, Rami Melhem, and Jun Yang. 2020. Enhancing Address Translations in Throughput Processors via Compression. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. ACM, New York, NY, USA, 191–204. https://doi.org/10.1145/3410463.3414633

[58] George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB Slice—a Low-Cost High-Speed Address Translation Mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. ACM, New York, NY, USA, 355–363. https://doi.org/10.1145/325164.325161

[59] Berthold Vöcking. 2003. How Asymmetry Helps Load Balancing. *Journal of the ACM* 50, 4 (jul 2003), 568–589. https://doi.org/10.1145/792538.792546

[60] VoltDB 2022. VoltDB Administrator's Guide, S2.3 - Configure Memory Management. https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[61] W. H. Wang, J.-L. Baer, and H. M. Levy. 1989. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (Jerusalem, Israel) *(ISCA '89)*. ACM, New York, NY, USA, 140–148. https://doi.org/10.1145/74925.74942

[62] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[63] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Antibes Juan-les-Pins, France) *(SIGMETRICS '16)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2896377.2901456

[64] Hongil Yoon and Gurindar S. Sohi. 2016. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*. IEEE, USA, 212–224. https://doi.org/10.1109/HPCA.2016.7446066

[65] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1810085.1810109

[66] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC '20)*. USENIX Association, USA, Article 57, 14 pages. https://doi.org/10.5555/3489146.3489203

[67] Sudarsun Kannan and Jaehyun Han. 2023. *oscarlab/mosaic-asplos23-artifacts: Mosaic ASPLOS'23 Artifacts*. https://doi.org/10.5281/zenodo.7709303