Mosaic Pages: Big TLB Reach with Small Pages

Jaehyun Han, *The University of North Carolina, Chapel Hill, NC, 27599-3175, USA*
Krishnan Gosakan, *AMD, Bengaluru, India*
William Kuszmaul, *Harvard University, Cambridge, MA, 02134, USA*
Ibrahim N. Mubarek, *Carnegie Mellon University, Pittsburgh, PA, 15213, USA*
Nirjhar Mukherjee, *Carnegie Mellon University, Pittsburgh, PA, 15213, USA*
Karthik Sriram, *Yale University, New Haven, CT, 06511, USA*
Guido Tagliavini, *Snowflake, San Mateo, CA, 99402, USA*
Evan West, *Stony Brook University, Stony Brook, NY, 11794-2424, USA*
Michael A. Bender, *Stony Brook University, Stony Brook, NY, 11794-2424, USA*
Abhishek Bhattacharjee, *Yale University, New Haven, CT, 06511, USA*
Alex Conway, *Cornell Tech, New York, New York, 10044, USA*
Martín Farach-Colton, *New York University, New York, NY, 11201, USA*
Jayneel Gandhi, *Meta, Menlo Park, CA, 94025, USA*
Rob Johnson, *VMware Research, Palo Alto, CA, 94304, USA*
Sudarsun Kannan, *Rutgers University, Piscataway, NJ, 08854-8019, USA*
Donald E. Porter, *The University of North Carolina, Chapel Hill, NC, 27599-3175, USA*

Abstract—This article 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. 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. Timing and area analyses on a commercial 28nm CMOS process indicate that the hashing required on the critical path can run at a maximum frequency of 4 GHz, indicating that a Mosaic TLB is unlikely to affect clock frequency.

Data-hungry applications, such as data and graph analytics, are often bottlenecked on the translation lookaside buffer (TLB). A typical TLB can cache only 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

Digital Object Identifier 10.1109/XXX.0000.0000000

cache translations for only about 8.6 MiB of physical memory at once. As a result, many modern applications report 20–30% overhead attributable to TLB misses [4, 6, 9], and some as high as 83% [1].

Given the challenges of building larger TLBs that meet tight CPU cycle times, a primary family of techniques to increase TLB reach leverage physical contiguity, including the use of ***huge pages***, segments, and opportunistic coalescing of contiguous entries. The downside of relying on physical contiguity is that 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. [10] 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. Other proposals accommodate limited amounts of discontiguity or "holes" in contiguous ranges [7, 8], but the performance gains are the result of the residual physical contiguity in the mappings.

This article 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 [3] for physical address compression and mitigating TLB conflicts. This article is based on our ASPLOS '23 paper [5].

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 of these h discrete translations to a page frame a ***Compressed Physical Frame Number (CPFN)***. By compressing translations, we can pack translations for several contiguous virtual pages into a single TLB entry, expanding TLB reach by $\log p / \log h$ without increasing the number of TLB entries. This article shows that we can increase reach by at least a factor of four using current TLB sizes.

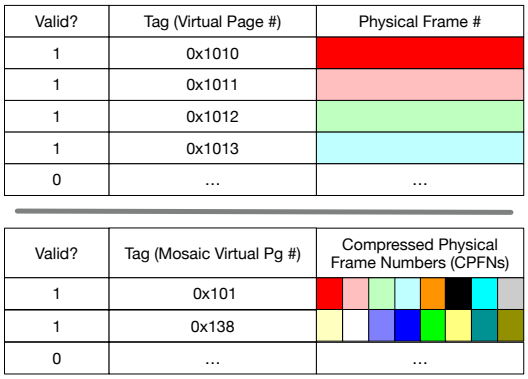

FIGURE 1: The top illustrates a traditional TLB mapping virtual addresses to physical page frames. Four contiguous virtual pages map to different physical addresses. The bottom depicts how a Mosaic TLB compresses the same pages into one entry, storing only the bucket and offset for each page.

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.

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. In any scheme that restricts mappings, the concern is forcing the eviction of a hotter page than an unconstrained mapping would. Smaller h decreases the size of TLB encodings but increases the chance of making a poor eviction choice during conflicts. This article shows that **it is possible to have a small h with comparable swapping costs.**

Our contributions. This article 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 and modify Linux to implement mosaic for anonymous, unshared pages.

Using this experimental infrastructure, we demonstrate that mosaic can indeed reduce TLB misses of

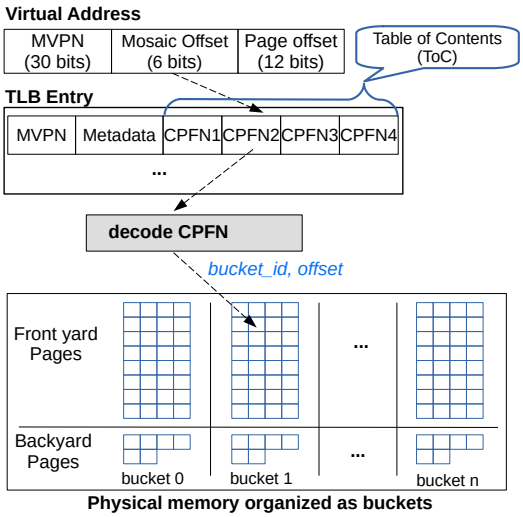

FIGURE 2: High-level Design of Mosaic Address Translation.

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 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 article demonstrates empirically that, under memory pressure, mosaic's swapping is comparable to an unconstrained page mapping. 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.

Mosaic Pages

A ***mosaic page*** is a large virtual page, composed of α virtually consecutive, but not necessarily physically contiguous, base pages (4 KiB). We say that α is the ***arity*** of a mosaic page. The key idea is to compress each translation such that translations for all α 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).

Mosaic pages increase the reach of the TLB by a factor of α by leveraging virtual locality. For example, consider current x86 TLBs, which use 36-bit physical frame numbers. If we use 8-bit CPFNs, then we can fit $\alpha = 4$ 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 $\alpha = 16$ or even larger.

Mosaic page tables map MVPNs to ToCs. Mosaic can use any page-table structure, such as radix trees or hash tables.

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. Thus, the CPFN needs to indicate only which of the h options was chosen by the page allocator.

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.

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.

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 the memory can be allocated (98% in our experiments) before seeing conflicts, with extremely high probability.

Mosaic allocates pages by using ***Iceberg hashing*** [3], 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.

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 buckets of some fixed size $f = \omega(\log\log p)$ (e.g., $f = \Theta(\log^2 \log p)$). The backyard also consists of s buckets, 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). 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).

When an item x is inserted, it first hashes to some bucket $h_0(x)$ in the front yard. If there is a free slot in $h_0(x)$, then the insertion uses that slot. Otherwise, if bucket $h_0(x)$ is full, then x is placed into the backyard. Elements in the backyard are assigned a bucket 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 buckets.

Evaluation

Mosaic Prototype

Our prototype implementation consists of three parts, mosaic TLB on gem5 full system simulator, mosaic page management in Linux, and hash function implementation in hardware. We use 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.

Does Mosaic Reduce TLB Misses?

We evaluate TLB behavior in gem5, varying 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. Our ASPLOS '23 paper describes these experimental parameters and results in more detail [5].

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.

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. When one considers sensitivity to arity, 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%. With an arity of 4, all CPFNs fit in a single unmodified x86 TLB entry. In terms of varying associativity, the results show that transitioning from a standard TLB to a mosaic TLB reduces TLB misses even further. For instance, a direct-mapped Mosaic-8 TLB outperforms a fully associative vanilla TLB in Graph500, BTree, and XSBench benchmarks. The results show that a Mosaic system could leverage more efficient, lower-associativity TLB designs.

Does Mosaic Reduce Memory Utilization?

We empirically measure the memory overhead, δ, caused by associativity conflict, and compare this to the memory utilization achieved by the default Linux virtual memory subsystem. Mosaic starts to have associativity conflicts once it hits a memory utilization of $1-\delta$. However, memory utilization can go beyond $1-\delta$ due to ghost pages. 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.

In Mosaic, the first conflict appears at around 98.03% utilization across all workloads, indicating that δ is roughly 2%. In contrast, we observed that vanilla Linux began swapping once memory utilization

reached 99.2% with the same configuration. 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 most of the available memory, and the overall memory overhead of Mosaic is less than 1%.

Does Mosaic Increase Swapping?

We run each workload with various 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`. When the workload slightly exceeds available memory, Mosaic swaps more than the default Linux allocator since Linux can utilize about 1% more memory than Mosaic. However, beyond this, Mosaic matches or surpasses Linux performance, sometimes by up to 29% in the best case. This may be because the associativity restrictions in Mosaic slightly perturb LRU's decisions, preventing the impact from known issues like cyclic memory references.

Is the Hardware Feasible?

We 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. 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.

Discussion

This section explores the impact of some of Mosaic's design choices.

Overlapping vs. Non-Overlapping Set Associativity

One lesson from Iceberg hashing is that, when reducing associativity, one can get better utilization with overlapping sets. In a traditional set-associative cache design, a given cache line maps to precisely one set, but the line may be placed in any way within that set. We call this a **non-overlapping** reduced associativity strategy. A well-understood failure mode for this approach is when frequently accessed memory locations are not evenly distributed across the sets, leading to under-utilization; in other words, with n-way set associativity, one can end up with $n+1$ hot lines in the same set, causing associativity conflicts. In practice, one may employ software-level strategies, such as page coloring, to ensure that frequently accessed memory locations are evenly distributed across the sets.

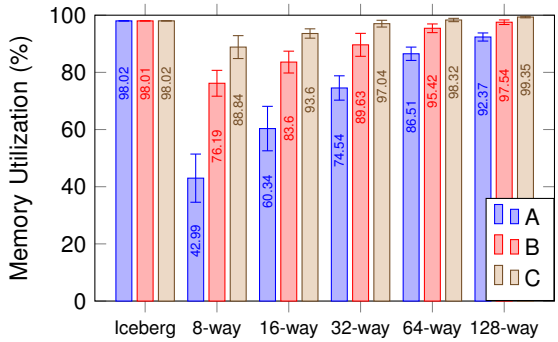

FIGURE 3: Memory utilization comparison (higher is better) between Iceberg hashing and traditional set-associative schemes with increasing numbers of ways.

A contribution of Iceberg hashing is a way to reduce associativity with good load balancing across the Iceberg buckets (i.e., the rough analog of sets in Mosaic). A key enabler of Iceberg's load balancing is that the buckets are **overlapping** for any given virtual address. Although Mosaic partitions physical memory at boot time into buckets, the mapping for any given virtual address spans multiple sets. So, for any group of $n+1$ "hot" virtual addresses, each address can be mapped to multiple, pages in different sets/buckets, avoiding associativity pathologies.

We present a study where we replace the Iceberg hash scheme with a simple set-associative page mapping scheme. Each page maps onto a set with an increasing number of ways, using the least significant bits of the virtual page number to select the set. We measure XSBench workload with sizes of 40 MiB (A), 400 MiB (B), and 4,200 MiB (C). 128-ways can be represented in the same number of bits as Iceberg's 104 choices.

Figure 3 shows that Mosaic's Iceberg hashing gives higher utilization for nearly all scenarios, except for the largest workload (C) with the largest set (128-way). This result illustrates that one is likely to get better empirical results with overlapping partition schemes. Moreover, Mosaic reduces the need for the application writer to work around associativity conflicts with techniques such as cache coloring.

LRU Precision

The original theory paper that described Iceberg [2] analyzed the impact on swapping using a page-eviction

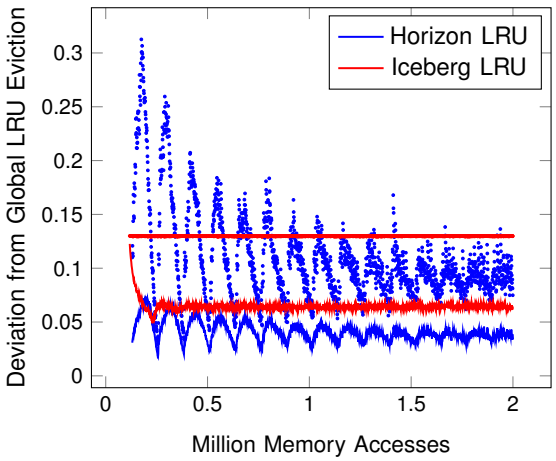

FIGURE 4: Comparison of page eviction choices under Iceberg LRU (red) and Horizon LRU (blue). The x axis shows eviction decisions over time, and the y axis shows the relative age of the eviction (lower is better). The lines indicate average relative age (and points indicate maximum relative age) over the last 1000 evictions.

algorithm called Iceberg LRU. Iceberg LRU evicts pages before memory is full in order to preserve invariants, in particular, that the oldest page is always evicted, even if it does not free a useful memory location. For our parameter regime, Iceberg LRU keeps 13% of memory free to ensure the oldest page is evicted and that there is a slot free for each new page. Unfortunately, it is generally undesirable to swap pages before memory is (nearly) full, since most use cases are are careful to avoid induced swapping by staying within the system's memory capacity.

A first optimization of Iceberg LRU is to mark the pages it wishes to evict as *evictable*, but only preform the eviction when space needs to be made for a page. This partially solves the issue, but keeping a large percentage of pages evictable at all times results in bad evictions and a high rate of swapping (this strategy is shown in red in Figure 4).

To address this issue, Mosaic uses ***Horizon LRU*** in which pages are marked evictable if they are older than the *horizon*. If an eviction must be performed and none of the eligible pages are evictable, then the horizon is advanced to the oldest eligible page. Unlike preforming evictions lazily, this change is a notable departure from the Iceberg theory. So, how does Horizon LRU perform as compared to fully associative (i.e. Global) LRU and Iceberg LRU? We note that, of course, LRU implementations are imprecise by nature — typically sampling accessed bits in page tables, so swapping schemes already tolerate some imprecision.

To understand this trade-off between LRU precision and overly eager swapping and to apply maximum pressure to Horizon LRU, we model a workload allocating unique virtual addresses without reuse. The results (for a memory size of 2^{17} pages) are shown in Figure 4. They indicate that Horizon LRU does suffer from exacerbated worst-case behavior at first (blue points above the bound of 13% oldest), but the system reaches a steady state where nearly all points are under this line. Moreover, Horizon LRU makes better swapping choices on average, at the cost of some noise from outliers. In our other experiments, this improved average case performance results in less swapping.

Conclusions

This article 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 pages are compatible with other techniques, such as huge pages, because any base page size can be mapped by TLB sub-entries.

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; Gosakan, Tagliavini, and Farach-Colton were at Rutgers; Kuszmaul was at MIT; and Conway and Gandhi were at VMware Research. 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, CCF-2106827, CCF-2106999, CCF-2118620, CCF-2118830, CCF-2118832, CCF-2118851, CCF-2119300, CNS-2154771, CNS-2231724, CCF-2247577, 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.

REFERENCES

1. Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, pages 237–248. ACM, June 2013.
2. 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. Paging and the address-translation problem. In *SPAA*, page 105–117, 2021.
3. Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. *J. ACM*, 2023.
4. Mel Gorman. Linux huge pages. https://lwn.net/Articles/375096/, 2010.
5. 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. Mosaic pages: Big TLB reach with small pages. In *ASPLOS*, page 433–448. ACM, 2023.
6. Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *OSDI*, pages 705–721, November 2016.
7. Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *ISCA*, pages 913–925, 2020.
8. Mark Swanson, Leigh Stoller, and John Carter. Increasing tlb reach using superpages backed by shadow memory. In *ISCA*, pages 204–213, 1998.
9. Michael M. Swift. Towards $O(1)$ memory. In *HotOS*, pages 7–11. ACM, 2017.
10. Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *USENIX ATC*, 2020.

Jaehyun Han is a Ph.D. candidate at the University of North Carolina at Chapel Hill. Contact him at jaehyun@cs.unc.edu.

Krishnan Gosakan is a Software Engineer at AMD. Contact him at krishnan.gosakan@gmail.com.

William Kuszmaul is a Postdoctoral Fellow at Harvard University. Contact him at williamkuszmaul@gmail.com.

Ibrahim N. Mubarek is a M.S. student at Carnegie Mellon University. Contact him at imubarek@alumni.cmu.edu.

Nirjhar Mukherjee is a Ph.D. student at Carnegie Mellon University. Contact him at nirjhar@cmu.edu.

Karthik Sriram is a Graduate Software Engineer at AMD. Contact him at mckarthik7@gmail.com.

Guido Tagliavini is a Software Engineer at Snowflake. Contact him at guido.tag@gmail.com.

Evan West is a PhD Candidate at Stony Brook University. Contact him at evan.ts.west@gmail.com.

Michael A. Bender is the John L. Hennessy Chaired Professor in Computer Science at Stony Brook University. Contact him at https://www.cs.stonybrook.edu/∼bender/ or bender@cs.stonybrook.edu.

Abhishek Bhattacharjee is a Professor of Computer Science at Yale University. Contact him at abhishek.bhattacharjee@yale.edu.

Alex Conway is an Assistant Professor at Cornell Tech. Contact him at https://ajhconway.com or me@ajhconway.com.

Martín Farach-Colton is the Leonard J. Shustek Professor of Computer Science at New York University. Contact him at martin@farach-colton.com.

Jayneel Gandhi is an Engineer at Meta. Contact him at jayneel@meta.com.

Rob Johnson is a Distinguished Engineer at VMware Research. Contact him at rob@robjohnson.io.

Sudarsun Kannan is an Assistant Professor at Rutgers University. Contact him at sudarsun.kannan@rutgers.edu.

Donald E. Porter is a Professor at the University of North Carolina at Chapel Hill. Contact him at porter@cs.unc.edu.