Pack Light, Walk Fast: Compact Page-Table Residency in Shared Caches

Samarth Narang[1,2] Aamer Jaleel[2] Brandon Reagen[1]
[1]New York University [2]NVIDIA

arXiv:submit/8387374 [cs.AR] 17 Nov 2026

Abstract

Hardware page-table walkers fetch descriptors through the cache hierarchy, causing page-table walks to compete directly with application data for shared-cache capacity. Retaining leaf page-table lines on chip avoids costly off-chip DRAM reads during address translation, yet as a workload's mapped footprint grows, those retained leaf lines displace application data and slow down cache-sensitive co-runners. We introduce LeafPack, a shared-cache organization that resolves this trade-off by compressing architectural page-table leaf lines in place within standard cache ways. By exploiting regularity in the physical frame numbers and attributes of neighboring descriptors, we pack up to sixteen 64-byte leaf lines into a single 80-byte slot inclusive of metadata. Our cache layout packs incoming leaf lines incrementally as walks fetch them from memory and reconstructs individual 64-byte lines on demand while leaving irregular lines uncompressed. This organization preserves both the in-memory page-table format and the hardware page-table walk algorithm, requiring only that walkers identify their requests as leaf accesses. To adapt page-table occupancy to changing workload demands, we pair this layout with a runtime set-dueling controller that arbitrates between compressed and uncompressed residency ceilings to balance translation coverage against application-data capacity. We evaluate LeafPack in full-system gem5 simulations of an Arm platform across both a small core and a server-class out-of-order core on workloads spanning one to eight cores. Across nine translation-heavy workloads, we improve geometric-mean harmonic speedup over an ordinary shared cache by 28.7% on the small core and 28.4% on the server-class core. Against an ordinary cache using SRRIP replacement, we improve geometric-mean harmonic speedup on these nine workloads by 18.7% and 17.7%. On twenty unseen held-out mixes evaluated after freezing the controller, LeafPack improves harmonic speedup over an ordinary cache on every mix, achieving geometric-mean gains of 21.4% on the small core and 18.5% on the server-class core.

1 Introduction

When a memory access misses in the TLB hierarchy, the hardware page-table walker resolves the translation through dependent descriptor accesses [15, 17]. Page-walk caches can satisfy or bypass upper-level accesses, shortening the walk, but accesses to leaf descriptors can still miss in the on-chip cache hierarchy and stall the walk on main memory. While upper-level descriptors cover wide spans of virtual address space with relatively few entries, the leaf level grows linearly with the mapped footprint, as mapping a dense 4 GiB region with standard 4 KiB pages requires more than one million eight-byte leaf descriptors totaling 8 MiB. When repeatedly accessed leaf lines outgrow the shared cache, walkers must fetch evicted lines from DRAM even if the upper levels remain on chip. Retaining those leaf lines in the cache avoids off-chip walk accesses, yet every line dedicated to page-table metadata displaces application data that cores also need.

Architects have attacked translation overhead from both the TLB side and the cache side. Enlarging TLBs, coalescing contiguous translations as in CoLT [57], and adding concurrent hardware walkers reduce walk frequency or overlap independent misses. The walks that still remain must fetch descriptors through the memory hierarchy, where cache replacement policies such as EWaH and xPTP [21, 53] reduce walk latency by preferentially retaining page-table lines in the data caches. Because these policies store leaf lines uncompressed, protecting a growing leaf footprint starves co-running applications of data capacity, and any leaf footprint that exceeds the cache size cannot fit without compression.

We observe that although page-table leaf lines are reused, the cache space occupied by them holds redundant information. Every eight-byte leaf descriptor combines a physical frame number with permissions and status bits, and neighboring descriptors frequently share identical attributes while referencing contiguous physical frames. Whenever the eight descriptors inside a 64-byte leaf line follow a regular progression, the line's first descriptor and stride determine the remaining seven entries, and neighboring leaf lines that share a common base frame allow their starting descriptors to be represented by small offsets. We exploit this spatial regularity inside the shared cache through compression to both accelerate standalone execution by retaining more active leaves than uncompressed protection allows and preserve application-data capacity by holding the same leaves in fewer ways.

We build on this insight to design *LeafPack*, a shared-cache organization that packs up to sixteen 64-byte leaf lines into an 80-byte slot inclusive of metadata. As hardware walkers fetch leaf lines from memory, LeafPack incrementally encodes regular lines into shared slots and reconstructs individual lines on demand while leaving irregular lines uncompressed. Pairing four payload ways with a fifth metadata way preserves the physical cache array, where per-line coherence bits maintain independent invalidation across sibling lines. We only require hardware walkers to tag their page-table and leaf requests, leaving the page-table format, walk algorithm, and operating-system TLB shootdowns unchanged.

LeafPack packs leaf lines incrementally as misses arrive while preserving per-line coherence. Because even a partially filled group occupies five physical ways, we charge all converted ways, including metadata and empty slots, when balancing translation coverage against application-data capacity.

LeafPack uses set dueling to choose between two compressed way ceilings and one uncompressed ceiling as translation and data

ISCA '27, Atlanta, Georgia, USA
2027. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

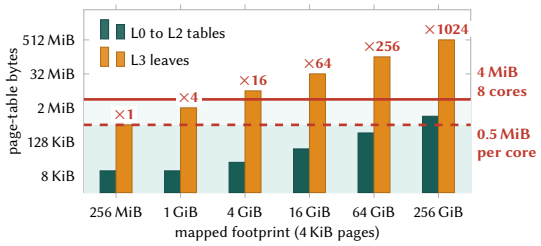


Figure 1: Page-table storage versus mapped footprint.

demands shift. Physical leader sets evaluate the compressed policies while shadow tags evaluate the uncompressed policy, and per-core miss weighting prevents high-traffic cores from dominating selection. Although one could combine prior page-table compression [50], shared-tag layouts [66], and adaptive cache partitioning [43, 60], compressing each page-table line on its own inside a shared-tag layout under our controller barely improves over uncompressed protection (Section 5.3). LeafPack instead shares a single base descriptor across neighboring leaf lines, packing far more page-table coverage into a bounded physical allocation.

We evaluate LeafPack in full-system Arm gem5 simulations on small and server cores across one to eight cores. Our baseline shared L2 stores page-table lines and application data without compression or page-table protection, using random replacement unless noted. On nine translation-heavy workloads, LeafPack improves geometric-mean harmonic speedup over the SRRIP baseline by 18.7% and 17.7% on the small and server cores. Across all 18 development workloads under random replacement, it gains 4.7% and 4.8% over the best static uncompressed configuration (including baseline fallback), or about 2.0% and 2.1% when we exclude standalone 4 GiB SysBench. Across twenty held-out mixes, our frozen controller improves every mix over the baseline with geometric-mean gains of 21.4% and 18.5%, and beats the best static uncompressed configuration by 2.9% and 2.1%.

This paper makes three contributions:

- **Compressed leaf residency.** We design a shared-cache organization that packs neighboring architectural leaf lines into five-way groups using a shared base descriptor while preserving per-line coherence and falling back to uncompressed storage when needed.
- **Adaptive way selection.** We design an online controller that extends set dueling with shadow tags and per-core miss weighting to select between compressed and uncompressed page-table way ceilings.
- **Full-system evaluation.** We evaluate LeafPack across single-program and multiprogrammed workloads to separate the gains of protection, compression, and online selection, compare against prior translation-caching designs, and examine held-out mixes and controller failure cases.

2 Background and Motivation

With 4 KiB pages and 8-byte descriptors, each 64-byte leaf line maps 32 KiB of virtual memory. While upper-level entries cover larger address ranges and fit inside small walk caches, reused leaf lines can outgrow the shared cache and force off-chip DRAM accesses (Fig. 1).

Translation competes with data. Prior architectures use the on-chip cache hierarchy in several ways to cut down the cost of page-table walks. Page-table-aware replacement policies such as EWaH and xPTP [21, 53] identify cache lines fetched by hardware walkers and retain them longer than application-data lines, while Victima [33] repurposes L2 cache blocks to hold clusters of completed translations evicted from the TLBs. Whether a mechanism retains raw page-table lines or synthesized translations, it consumes physical cache ways that would otherwise hold application data. Its net benefit therefore depends on whether the off-chip walk accesses it avoids outweigh the additional application-data misses caused by shrinking the effective data cache, especially when a translation-heavy workload shares the cache with a data-sensitive co-runner.

Regularity creates a capacity opportunity. The internal structure of page-table leaf lines offers a natural way to break this capacity trade-off through compression. Because neighboring eight-byte leaf descriptors regularly share permission and status flags while referencing contiguous or nearby physical frames, multiple 64-byte leaf lines can be represented compactly using shared base descriptors and small frame offsets (Fig. 2). Turning this regularity into real shared-cache capacity savings requires packing leaf lines incrementally as walkers fetch them on demand, maintaining per-line coherence, falling back to uncompressed storage for irregular mappings, and returning emptied ways to application data.

3 LeafPack

A *leaf line* is a 64-byte cache line containing eight 8-byte leaf page-table entries (PTEs) [8, 73]. LeafPack compresses and combines these lines to increase the shared L2's effective page-table capacity.

3.1 Leaf Compression

Both LeafPack formats (Fig. 2) exploit regularity within a leaf line. When all eight PTEs are valid page descriptors that map consecutive frames $f, f+1, \ldots, f+7$ and match in every bit outside the frame field (bits $[63:48]$ and $[11:0]$, including the access flag, dirty state, and software-defined bits), storing the first 8-byte PTE verbatim determines the remaining seven and reduces the 64-byte leaf line to an 8-byte descriptor. Any other line, including one with an invalid PTE, stays uncompressed.

Packed format. Packed stores the first descriptors of up to eight leaf lines in a 64-byte payload. With 16 bytes of metadata, this payload forms an 80-byte logical *slot*, representing 512 bytes at 6.4× compression when full. Each line retains its own starting frame and attributes, so Packed requires no contiguity across line boundaries.

Dense format. Dense represents up to sixteen leaf lines using one 64-bit base descriptor, a 16-bit intra-line stride, and sixteen signed 20-bit frame offsets. The base stores the first PTE of the first line placed in the slot, each offset gives a line's starting frame relative to that base ($\pm 2^{19}$ frames, or ±2 GiB), and the stride sets each line's remaining frames (fixed to one in our evaluation). A full Dense slot represents 1,024 bytes in the same 80-byte footprint for 12.8× compression, and its offsets can bridge gaps between physical

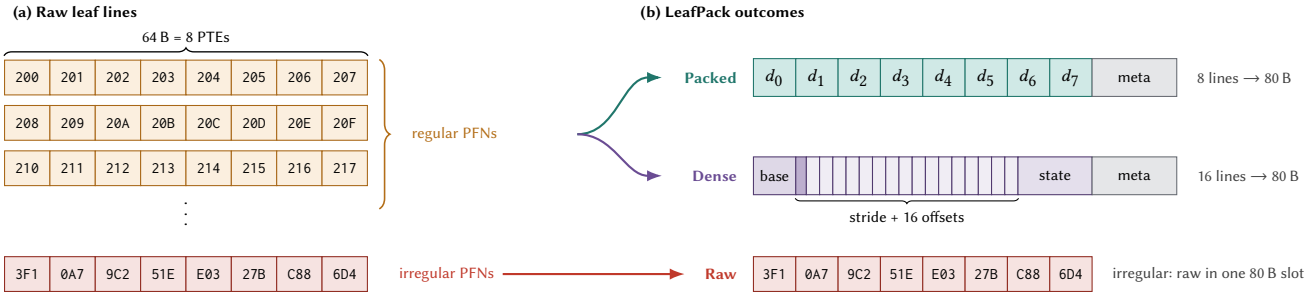


Figure 2: LeafPack leaf-line formats. (a) Regular and irregular physical frame numbers (PFNs). (b) Packed, Dense, and raw slots.

allocations as long as each line is internally contiguous and shares the base's non-frame bits.

3.2 Cache Organization

Physical storage. LeafPack distributes four logical slots across a *five-way group* within one cache set: four ways hold 64-byte payloads, and the fifth holds four 16-byte metadata records. This arrangement preserves the physical cache-array and bus widths.

A runtime policy selects compressed or uncompressed storage and a per-set way ceiling W for the page-table class (Section 3.5). The ceiling reserves no capacity: application data can use any way not held by page tables. A converted group counts as five page-table ways even when partially full, since its shared metadata way prevents application data from using empty slots. Section 5.1 describes partial groups used in static experiments.

Regions and slot tags. A *region* is an aligned 1 KiB range of physical memory holding PTEs, distinct from the physical frames those PTEs map. A compressed slot holds lines from a single region at fixed positions indexed by address bits $[9:6]$, where Dense stores position p's offset in field p and Packed stores position p's descriptor in word p mod 8 so that each Packed slot covers one 512-byte half of the region. Any subset of those positions can be valid at a time, and while a region may span multiple slots, each line position is valid in at most one slot.

The address bits above the 1 KiB boundary form the *region prefix*, which identifies the region during lookup (Section 3.3). Each slot's 61-bit tag record contains this prefix and sixteen per-line valid bits. Each valid bit indicates cache residency of the corresponding line, independently of PTE validity. These records reside in the tag array, outside the 80-byte slot footprint. The slot's 16-byte metadata record holds per-line coherence state and per-slot replacement state, using 71 of its 128 bits. A three-bit *group bitmap* per set identifies converted five-way groups.

3.3 Cache Access and Replacement

Lookup. An L2 lookup checks the line's normal set for uncompressed storage and its *packed set* for either compressed format. The packed-set index uses only the region prefix, excluding the low ten address bits, so all sixteen line positions map to the same set. Address bits $[9:6]$ select the line position while bits $[5:0]$ select the byte.

The group bitmap restricts slot probes to converted groups. A compressed hit requires a matching region prefix and a valid bit of one for the requested line position. The cache selects that line's stored descriptor or frame offset and reconstructs the original 64-byte line, adding two cycles for Packed or three for Dense. Unconverted sets retain the standard L2 path with no added latency.

Fill and allocation. L2 walk misses fetch individual 64-byte lines from memory. Under a compressed policy, the first eligible leaf line in a region opens a Dense slot and sets its base descriptor. Later eligible lines from that region join the Dense slot whenever their non-frame bits match the base and their starting frame fits the 20-bit offset range. If a line cannot join the region's Dense slot, it goes to a Packed slot for that region instead, either joining one with space or opening a new one. Upper-level page-table lines and leaves with irregular or invalid PTEs stay uncompressed, taking one slot in their normal set under a compressed ceiling or a regular way under the uncompressed ceiling.

A fill needing a new slot first uses an open slot in a converted group. If none is available and the ceiling permits, the cache converts another five-way group, evicting its application data. At the ceiling, it replaces a slot using bimodal re-reference interval prediction (BRRIP) [30]. Application data in unconverted ways uses the baseline replacement policy.

Replacement and group release. Because BRRIP tracks a single re-reference value per slot rather than per line, evicting a victim slot evicts every leaf line it holds. Once evictions or invalidations clear the last valid line from a group, the cache clears its group bitmap bit and returns all five ways to application data. Lowering the way ceiling can also reclaim excess groups (Section 3.5).

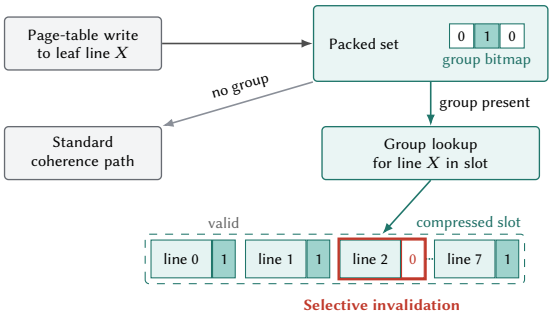


Figure 3: Coherence handling for compressed lines.

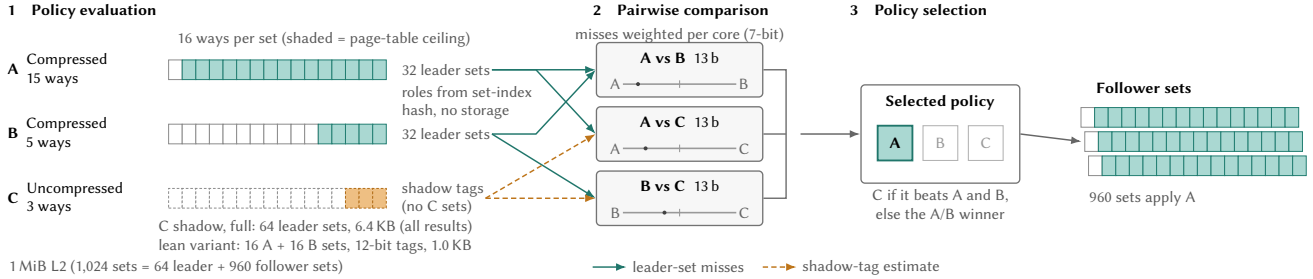


Figure 4: Adaptive way selection.

3.4 Coherence and Invalidation

LeafPack caches architectural page-table lines without changing their layout in memory and participates in standard cache coherence [69]. Coherence requests locate a compressed copy using the packed set's group bitmap, region prefix, and line position (Fig. 3). An invalidation clears only that line's valid bit, leaving other lines in the slot accessible.

When a write to a compressed line still satisfies its slot's format, the cache updates the entry in place. Otherwise, the cache expands that line to uncompressed form inside the slot and evicts the slot's remaining lines, or writes the updated line back to memory if another line in the slot has an outstanding miss. Even when the runtime policy changes, lookups always check both the normal and packed sets so that every resident line remains reachable at a single location. Moreover, the cache never evicts a line that has an outstanding miss and executes every fill, re-encoding, or group conversion atomically on its set. Operating-system TLB shootdowns flush the TLBs as usual [46] and require no LeafPack invalidation.

3.5 Adaptive Way Selection

LeafPack uses set dueling [59] to select among three policies (Fig. 4): a 15-way compressed ceiling for large active leaf sets, a 5-way compressed ceiling to protect cache-sensitive co-runners, and a 3-way uncompressed ceiling for small leaf sets.

Policy comparison. Leader sets are sampled sets running candidate policies; we assign 32 physical leaders to each compressed policy. Each of these 64 leaders also has *shadow tags*, tag and state entries that estimate hits under the uncompressed policy without storing data. They track up to 3 page-table entries (under SRRIP) and 16 data entries (under the baseline's data replacement policy) per set while capping the combined total at 16 valid entries to match a physical set, avoiding uncompressed leader sets that penalize translation-heavy walks.

Three saturating counters compare demand misses pairwise, weighting each miss inversely by its core's L2 access rate so high-traffic cores cannot dominate quieter co-runners. The remaining *follower sets* choose the uncompressed policy when it beats both compressed policies, and the winning compressed policy otherwise.

Warm-up and probe modes. To let compressed leaders accumulate neighboring leaf lines, a warm-up gate freezes the two counters comparing compressed policies against shadow tags until the compressed leaders have observed 2^{15} accesses. Followers initially choose only between the compressed policies.

After warm-up, probe modes reduce capacity spent on losing policies. When the uncompressed policy beats both compressed policies by a quarter of the counter range, *uncompressed probing* switches 30 of each compressed policy's 32 leaders to uncompressed storage, retaining two per policy to detect phase changes.

When one compressed policy beats the other by a quarter of the counter range and also beats the uncompressed policy, *compressed probing* switches 30 losing leaders to the winner. The winner's 32 original leaders continue comparisons against their shadow tags, while two matched leader pairs compare the compressed policies. In either probe mode, if the winning margin falls below half the entry threshold or the follower policy changes, the controller exits probe mode, restores all 64 leaders to their home policies, and suppresses counter updates for 4,096 leader accesses while they refill.

Changing the ceiling. A lower ceiling limits new page-table insertions to one five-way group under the 5-way policy or three uncompressed ways under the 3-way policy. Excess converted groups remain searchable for walk hits, avoiding an immediate cache-wide flush. When an application-data miss finds all unconverted ways occupied, the cache reclaims one excess group by evicting its resident lines (up to 64) and returning its five ways to data. If the write buffer fills before all lines are evicted, the group stays converted and subsequent data misses finish reclaiming it.

3.6 Hardware Overhead

Controller storage. Hashing the set index assigns leader roles without lookup storage. At each epoch's end, the controller updates per-core 7-bit miss weights from per-core access counters, off the critical path. The three 13-bit selection counters, per-core weights, epoch counters, and mode registers require 322 bits for eight cores or 147 bits for one core. Shadow tags require 6.4 KB; a lean variant using 12-bit partial tags and 16 leaders per compressed policy reduces this to 1.0 KB while matching performance within 0.3%.

Implementation cost. To estimate hardware area, we model the 22 nm L2 arrays in CACTI [9] and synthesize our pipelined encoder, decoder, and slot-selection RTL with Synopsys Design Compiler in a 45 nm library. Each L2 uses a single encoder and decoder that accept one operation per cycle and, together with the slot-selection logic, take 24.4k NAND2-equivalent gates (18,712 μm^2 at 45 nm). Scaling that logic to 22 nm by a factor of 0.24 to 0.35 (accounting for $(22/45)^2$ feature scaling and interconnect overhead) adds 0.38 to 0.56% to the area of a 1 MiB L2 cache, with each stage fitting inside a standard 64-bit ALU delay to support one-cycle fill encoding and two- or three-cycle decoding.

At 1 MiB and 4 MiB, the added tag and metadata bits increase L2 array area by 1.20% and 1.74% (leaving array access time unchanged at 1 MiB and adding 1.8% at 4 MiB), the 6.4 KB shadow tag array adds 3.01% and 0.85% (or 0.80% and 0.23% for the lean 1.0 KB variant), and the controller, transition logic, multiplexers, and staging buffer add an estimated 0.2 to 0.5% and 0.07 to 0.14%. Altogether, LeafPack adds 4.8 to 5.3% to the area of a 1 MiB L2 and 2.8 to 2.9% to a 4 MiB L2 (or 2.6 to 3.1% and 2.1 to 2.3% with lean shadow tags) while increasing leakage power by 3.2% and 2.4%, excluding request sideband wires and place-and-route overhead.

4 Implementation and Methodology

Implementation and timing. We implement LeafPack in roughly 2,000 lines of C++ in gem5 [19, 40] (version 25.1) using its compressed-cache framework [26], which supports multiple independently tagged logical lines per physical entry. We extend the shared L2 with the compressor, packed-set tag store, and set-dueling controller from Section 3 while leaving the TLBs, walk algorithm, and coherence protocol unmodified (walkers only tag page-table and leaf requests). We charge one cycle for fill encoding and add two or three cycles to L2 hits for packed or dense decoding, probing normal and packed sets in parallel assuming a banked tag array without port contention, and test a single-ported array in a separate model (Section 5.6). We verify that disabling LeafPack reproduces unmodified gem5 statistics on every checkpoint, and we validate our tag store and coherence paths using runtime assertions, a 23.4M-operation reference-cache stress tester, and full-system page-remapping tests.

System and measurement. We run full-system Arm simulations under Linux 4.18 with 4 KiB base pages across two 4 GHz out-of-order core models, a small core (128-entry ROB with 12 L1 and 48 L2 MSHRs) and a server core (320-entry ROB with 48 L1 and 96 L2 MSHRs). Each core has private 32 KiB two-way L1 instruction and data caches, 64-entry L1 TLBs, a 1,280-entry five-way L2 TLB (caching upper-level partial translations so hits fetch only the leaf line), and up to 32 data walkers plus one instruction walker. The cores share a mostly exclusive 16-way L2 last-level cache with a 12-cycle access latency, a stride prefetcher, and 0.5 MiB of capacity per core (1, 2, and 4 MiB for two, four, and eight cores, while keeping 1 MiB for single programs to match the two-program pairs), backed by a dual-rank DDR3-1600 channel. We fast-forward on an atomic CPU to profiled phases and restore the same full-system checkpoint on the out-of-order CPUs for every configuration, warming up for 40 million cycles before simulating twenty 20-million-cycle intervals and reporting metrics over the last nineteen (380 million cycles, where including the first interval changes geometric-mean speedup by at most 0.2%).

Workloads and validation. Table 1 summarizes our unmodified benchmarks, where the *translation* benchmarks allocate leaf page tables that exceed the 0.5 MiB per-core L2 budget by 9.0× to 17.0× (measuring total allocated leaf storage rather than the active subset reused in a single phase), while the *co-runner* benchmarks have small leaf footprints and stress L2 data capacity. We run these programs alone and pinned one per core in two-, four-, and eight-program mixes,[1] separating them into an 18-workload development set (nine translation-heavy headline workloads and nine check workloads) used to design the controller and 20 randomly sampled held-out mixes (14 pairs and six four-program mixes) evaluated only after freezing the simulator binary.

Table 1: Benchmark page-table footprints.

Program	Suite	Input	Mapped (GiB)	Leaves (MiB)	Ratio (×)	Role
bc	GAP [12]	U24 graph	2.46	4.94	9.9	translation
pr	GAP [12]	U24 graph	2.25	4.52	9.0	translation
sssp	GAP [12]	U24 graph	4.22	8.48	17.0	translation
bfs	GAP [12]	U24 graph	2.21	4.43	8.9	moderate
XSBench	XSBench [76]	large grid	3.76	7.53	15.1	translation
SysBench	SysBench [38]	4 GiB read	4.00	8.02	16.0	translation
SysBench	SysBench [38]	512 KiB read	0.003	0.02	0.05	co-runner
omnetpp	CPU2026 [42]	reference	0.03	0.07	0.1	co-runner
gcc	CPU2026 [42]	reference	0.33	0.73	1.5	co-runner

Baselines and metrics. We evaluate the baseline cache with random replacement and, where noted, with 3-bit SRRIP or LRU. Both static uncompressed protection and LeafPack use the matching baseline replacement policy for application data while managing page-table lines with BRRIP [30] under a per-set way ceiling W. Under identical cache geometries, we compare the baseline against static uncompressed configurations ($W \in \{3, 5, 10, 15\}$), static compressed configurations ($W \in \{4, 5, 10, 15\}$, plus compressed $W = 3$ and uncompressed $W = 4$ on pr for the capacity study in Section 5.1), LeafPack, and CoLT [57] TLB coalescing. We also implement Victima [33], EWaH [53], and xPTP [21] inside our shared L2 cache. In our Victima port, whenever its walk-cost predictor detects high translation pressure, the cache stores eight-translation clusters in dedicated L2 blocks under TLB-aware SRRIP and probes the L2 in parallel to skip page-table walks on a cluster hit. We do not charge Victima's 64 KiB predictor table or cluster headers against L2 capacity, and we report its best per-set way cap from $N \in \{4, 8, 16\}$ on each workload. To model EWaH and xPTP, we add a priority bit to each L2 cache line with no access-latency penalty. Under EWaH, the cache protects page-table walker fills from eviction 99% of the time while the requesting core experiences frequent TLB misses, and it never allows one core's data accesses to evict another core's protected page-table lines. Under xPTP, which relies on LRU recency order, the cache evicts the LRU data line whenever one sits within eight positions of the LRU tail (falling back to the true LRU line otherwise) and turns off page-table priority when translation pressure is low.

We measure single-program throughput using IPC and multiprogrammed throughput using harmonic speedup [41] relative to the same mix on the baseline, $\mathrm{HS} = n/\sum_i (\mathrm{IPC}_i^{\mathrm{base}}/\mathrm{IPC}_i^{\mathrm{config}})$. Harmonic speedup reduces to the IPC ratio for one program and penalizes any policy that accelerates one program by starving a co-runner of data capacity. To explain the resulting trends, we also track page-walk misses, data misses, and page-table cache occupancy.

[1] Mix-1 combines bc, two pr, and 4 GiB SysBench. Mix-2 combines bc, XSBench, sssp, 4 GiB SysBench, two omnetpp, and two gcc. Mix-3 combines bc, bfs, pr, sssp, XSBench, 4 GiB SysBench, and two omnetpp.

5 Evaluation

Unless noted otherwise, paired numbers report the small and server cores using the cache sizes from Section 4.

Interpreting the comparisons. We evaluate LeafPack against three reference points to separate its total speedup into the benefit of protecting page-table lines, the benefit of compressing them, and the cost of online policy selection.

- **Baseline.** We use the shared L2 cache defined in the introduction to measure LeafPack's overall harmonic speedup (*HS*).
- **Best static uncompressed.** We pick the highest-performing configuration offline for each workload among the baseline and static uncompressed protection ($W \in \{3, 5, 10, 15\}$) to isolate the gain of leaf compression beyond page-table protection alone.
- **Best static compressed.** We pick the highest-performing static compressed configuration offline for each workload ($W \in \{4, 5, 10, 15\}$) to measure the cost of restricting our runtime controller to three candidate policies and selecting among them online.

5.1 Capacity Savings for Single Programs

To assess the headroom available from leaf compression before evaluating our runtime controller, we first test static way ceilings. While our adaptive controller is restricted to the three candidate policies from Section 3.5 (five or fifteen compressed ways, or three uncompressed ways), we also allow the static sweep to form *partial* groups in which one way holds metadata and the remaining ways hold compressed payloads (forming three 80-byte slots across four ways at W=4 with 16 metadata bytes unused, or two slots across three ways at W=3 with 32 bytes unused). Our reported compressed occupancy charges every converted way, including metadata, unused bytes, and empty slots, while uncompressed occupancy counts resident 64-byte lines.

As shown in Fig. 5, compressing leaf lines matches the IPC of the best static uncompressed configuration while consuming 50 to 73% less physical L2 capacity (saving 73% on bc and 4 GiB SysBench, 64% on pr, 60% on XSBench, and 50% on sssp). Under an equal five-way ceiling, LeafPack improves IPC whenever the active leaf working set overflows the uncompressed ways, gaining 22.0% on bc and 9.9% on XSBench. On bc, four compressed ways already match fifteen uncompressed ways on both the small and server cores, so additional page-table ways provide little standalone benefit and instead remain available to co-runners. By contrast, on 4 GiB SysBench, the active 8 MiB leaf working set is too large to fit in the 1 MiB L2 even at fifteen uncompressed ways, so leaf compression continues to expand translation coverage all the way to W=15 and raises IPC by 1.65× over fifteen uncompressed ways (1.71× over the baseline).

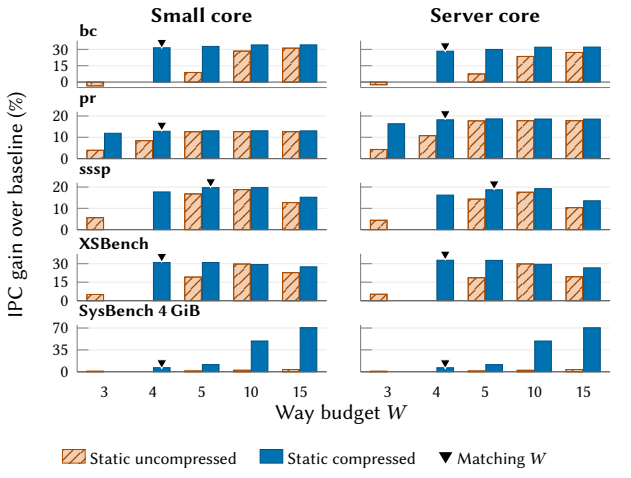


Figure 5: Single-program capacity savings. Triangles mark the smallest compressed W within 0.1% of the best uncompressed IPC.

5.2 Adaptive Controller

Figure 6a evaluates our frozen runtime controller across all 18 development workloads. On the nine translation-heavy headline workloads (left), LeafPack improves geometric-mean harmonic speedup over the baseline by 28.7% on the small core and 28.4% on the server core, outperforming the best static uncompressed configuration on both cores and staying within 1% of best static compressed on every run except single-program bc and server-core Mix-2. On the nine check workloads (right), our controller never falls more than 0.5% below the best static uncompressed configuration except on the two co-runs with gcc (Section 5.7).

Adaptation and oracle gaps. Figure 7 illustrates how our runtime controller converges on each of the three candidate policies across three representative workloads. When two instances of bc share the cache (Fig. 7a), the 15-way compressed policy A beats the 5-way policy B by 7.5 to 9.7% in every interval and the controller tracks policy A from the start. When bc instead co-runs with data-hungry omnetpp (Fig. 7b), policy B beats policy A by 15.4% by preserving data capacity for omnetpp, and the controller follows policy B across the window. When omnetpp runs alone (Fig. 7c), its small leaf footprint fits inside 3 uncompressed ways so policy C beats both compressed policies by 2%, and the controller selects policy C and enters probe mode within three intervals so only four physical leader sets continue running the compressed policies.

The controller also adapts when shared-cache demand shifts mid-run, switching follower sets between policies A and B across pr's traversal phases on pr+omnetpp. To test a sharper phase change where no single static policy is optimal for the entire run, we pause the data-sensitive co-runner in 4 GiB+512 KiB SysBench for ten intervals before resuming it. The controller shifts from policy C to the 15-way policy A while 4 GiB SysBench runs alone and then drops to the 5-way policy B once the co-runner resumes, outperforming every static policy over the full run (1.124 versus 1.111 harmonic speedup, while trailing the best static choice by 2.3% and 3.4% in the same test on pr+gcc).

Among our headline workloads, standalone bc and server-core Mix-2 show the two largest gaps to the best static compressed configuration for different reasons. On standalone bc, nearly tied leader-set miss counts cause the controller to pick five compressed ways, leaving a 1.2 to 1.9% gap below fifteen ways, whereas on server-core Mix-2 the best static setting uses a four-way partial group outside our candidate set, where the best candidate reaches 0.985 and our controller reaches 0.982.

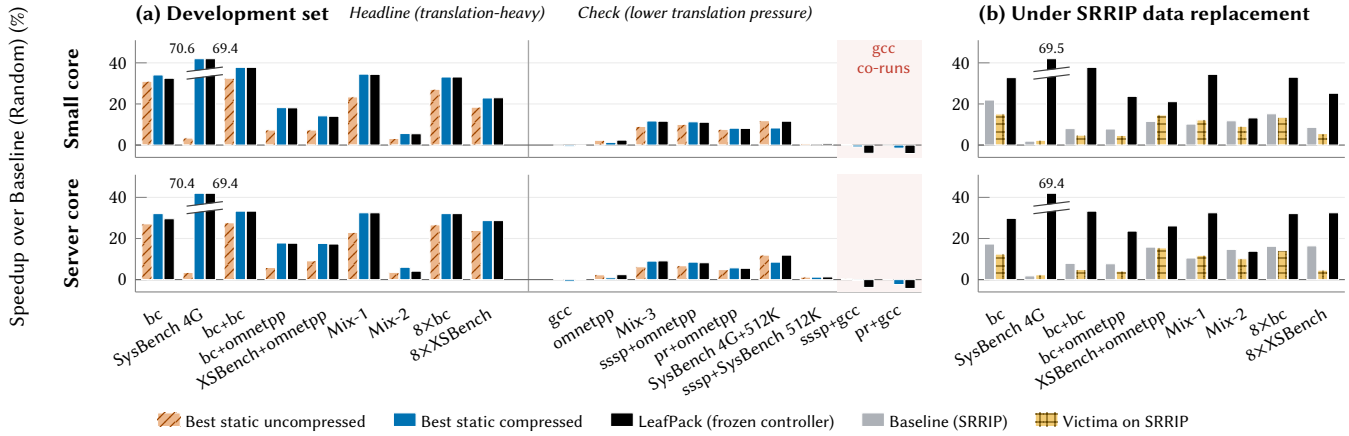


Figure 6: Harmonic speedup gain over our random-replacement baseline across the development set (a) and under SRRIP data replacement (b) on the small (top) and server (bottom) cores. LeafPack trails the SRRIP baseline only on server-core Mix-2.

Controller ablations. We next ablate our controller's design choices to see how each mechanism contributes. First, evaluating the 3-way uncompressed policy in shadow tags rather than dedicated physical leader sets avoids forcing translation-heavy accesses through sets whose three ways cannot hold a large leaf working set, gaining 2.0% on bc+bc. Second, entering compressed probe mode cuts the ongoing miss cost of keeping all 32 leader sets on a losing compressed ceiling, lifting 4 GiB SysBench by 2.0% and the omnetpp pairs by 0.3 to 0.9%. Finally, because compressed slots fill incrementally over time, our warm-up gate prevents early cold misses in compressed leader sets from locking follower sets onto the uncompressed policy during warm-up, raising harmonic speedup on server-core 8×XSBench from 0.992 to 1.287.

Selection accuracy and fairness. Across the full cache, our per-core weighted miss metric picks the candidate with the highest harmonic speedup in 35 of 36 workload-core cases (with the sole exception on server-core pr+omnetpp being a 0.03% tie), though leader-set sampling can cost 3 to 4% in harmonic speedup when candidates differ by under 2% in weighted misses. To separate online selection loss from the cost of restricting the controller to three candidates, we compare against all eight static configurations from Section 4. Across the 18 development workloads on both cores, the best of our three candidates reaches 1.000 and 0.999 of the best among all eight

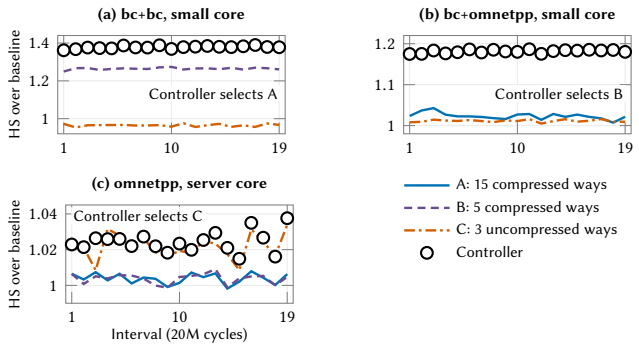


Figure 7: Runtime policy adaptation.

static configurations (candidate-set loss), and our online controller reaches 0.994 of its best candidate (selection loss), with most of that gap coming from the two gcc pairs (0.964 and 0.960, where candidate C is the best candidate). Outside the development set, fragmented XSBench experiences both selection and candidate-set loss (reaching 0.911 and 0.919 of its best candidate, while its best candidate reaches 0.946 and 0.928 of the 10-way static ceiling in Section 5.6). Finally, even when a multiprogrammed mix improves in overall harmonic speedup, individual co-runners within the mix can still experience a slowdown [37] when page-table ways displace their data (slowing by 0.9 to 16% in mixes that improve overall, and by up to 24% for gcc when paired with sssp, as analyzed in Section 5.7).

5.3 Where the Gain Comes From

Across all 18 development workloads, LeafPack improves harmonic speedup by 15.7% and 15.1% over the baseline. While protecting page-table lines accounts for most of that gain, LeafPack still leads the best static uncompressed configuration by 4.7% and 4.8%, which we break down in the rest of this section.

Figure 8a separates LeafPack's speedup into the gain of protecting page-table lines (captured by the best static uncompressed configuration) and the additional gain of compressing them. Protection accounts for most of the gain on standalone bc, bc+bc, and 8×bc, where no cache-sensitive co-runner competes for data ways, whereas compression accounts for the bulk of the gain when the active leaf set exceeds fifteen uncompressed ways (+63.7% on 4 GiB SysBench) or when retaining raw leaves displaces co-runner data (+10.0% on bc+omnetpp), while for gcc and omnetpp alone our controller selects the 3-way uncompressed policy C.

Figure 8b shows how our controller divides L2 capacity. On bc+omnetpp, the controller caps page tables at 31% of the L2 so application data retains 68%, whereas on bc+bc and 8×bc it allows page tables to occupy 94% of the L2 to capture the combined leaf set. Over 99% of compressed leaf lines use our sixteen-line dense format because its 20-bit offsets span ±2 GiB of physical memory. Depending on how many neighboring leaf lines a workload touches before eviction, each occupied slot holds an average of 3.0 lines on the

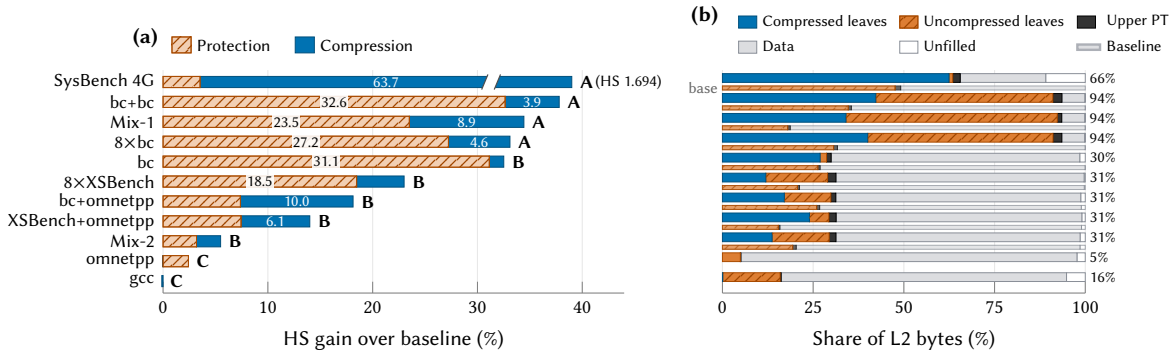


Figure 8: Speedup breakdown and L2 occupancy on the small core, showing (a) gains from protection and from compression beyond the best static uncompressed configuration (with A, B, and C marking the selected policy) and (b) L2 occupancy under our controller (upper) versus the baseline (lower) with page-table shares labeled.

eight-core mixes to 14.9 lines on 4 GiB SysBench, giving a median compression density of 4.7× over raw storage (2.1× to 10.4×) in static 15-way runs. The achieved density remains below the formats' maximum compression ratios of 12.8× and 6.4× because occupied slots are only partially filled. In Fig. 8b, *Unfilled* denotes cache capacity holding no valid lines, which consists mainly of empty slots inside converted groups that application data cannot use (10.9% of the L2 on 4 GiB SysBench, not counted in the page-table shares at right).

High compression density alone does not predict speedup, however, as standalone bc packs 6.5 lines per slot and holds 32% of its allocated leaf footprint yet gains only 1.0% over uncompressed protection because 15 uncompressed ways already hold its active leaf set. Instead, speedup over the best uncompressed configuration tracks the net reduction in L2 misses per 1,000 instructions (page-walk misses saved minus extra application-data misses), reaching a correlation of 0.72 without SysBench and 0.98 with SysBench included (versus 0.51 for slot occupancy alone).

LeafPack versus composing existing techniques. A natural alternative to LeafPack is to combine a page-table codec with an existing superblock compressed-cache layout [66] under an adaptive retention policy. Our packed-only configuration evaluates this combination by encoding each leaf line independently with LeafPack's codec in the same superblock slots under the same controller and decode latencies, so that only the cross-line format differs from full LeafPack. We also test a general-purpose per-line BDI codec [55] in the same layout. Across all 18 development workloads on both cores, the packed-only combination reaches 1.109 and 1.103 harmonic speedup over the baseline, sitting just 0.5% above the best static uncompressed configuration (1.104 and 1.098) and falling below it once we exclude 4 GiB SysBench, whereas full LeafPack reaches 1.157 and 1.151 for an additional 4.3% and 4.4% gain. That gap comes from sharing a base descriptor across neighboring leaf lines, which no per-line codec can do and which raises packing density on our checkpoint page tables from 5.1–6.0 leaf lines per physical way under per-line packing (or 6.0–7.0 under idealized within-line packing) up to 8.3–11.3 under LeafPack. Because per-line BDI fits only about two leaf lines per slot, it trails even the best static uncompressed configuration (0.970 and 0.969) while full LeafPack leads it by 7.9% and 8.1%, showing that our page-table format rather than generic compression drives the speedup. Outside the development set, fragmented XSBench is the sole exception, as the two simpler formats reach 1.14 to 1.18 versus 1.05 for full LeafPack in two of the three allocator histories (Section 5.6).

Remaining headroom. To bound how much further headroom remains, we construct an ideal leaf oracle that serves every leaf page-table read at L2-hit latency while consuming zero L2 cache capacity. Across eleven development workloads, LeafPack captures 77.8% (small core) and 78.8% (server core) of this ideal oracle's geometric-mean gain over the baseline, reaching 83 to 95% on single programs and homogeneous mixes and 70 to 80% on the omnetpp pairs, while on pr+gcc the ideal oracle gains 5.5 to 8.7% whereas LeafPack loses about 4% due to data-capacity contention.

5.4 Held-Out Workloads

Table 2 evaluates our frozen controller across all twenty unseen held-out mixes. LeafPack improves harmonic speedup over the baseline across all twenty mixes on both cores, reaching geometric means of 1.214 and 1.185 (and at least 1.026 on every mix) while keeping each individual co-runner at or above 0.95 of its baseline IPC on two-program pairs and 0.93 on four-program mixes. Against the best static uncompressed configuration, our controller achieves geometric-mean ratios of 1.029 and 1.021 (minimum 0.946 and 0.954 on bc+sssp), while under SRRIP data replacement the twenty mixes gain 9.9% and 7.4% over the SRRIP baseline and 2.8% and 1.8% over the best static uncompressed configuration on SRRIP.

To test generalization to applications outside our benchmark pool, the bottom of Table 2 evaluates the DBx1000 in-memory database [81] running YCSB [23], the Olden mst pointer-chasing benchmark [62], and the SPEC lbm stencil alone and in pairs, with all workloads and pass criteria fixed before simulation. Across these six workloads on both cores, LeafPack never trails the baseline by more than 0.1% or the best static uncompressed configuration by more than 1%, while accelerating DBx1000 by 15.4% and 19.6% over the baseline (8.4% and 10.2% beyond uncompressed protection).

Where compression matters. The best static uncompressed configuration is a strong reference point because picking the best fixed uncompressed ceiling offline per workload bounds any single fixed ceiling, and while it does not bound priority-based or TLB-caching

protection by construction, EWaH, xPTP, and Victima at its best way cap never exceed it by more than 0.4%, 0.3%, and 0.1% on any run (each under its own data replacement policy). Compression opens up a trade-off that this offline oracle cannot match, either delivering the same translation coverage in 50 to 73% less page-table capacity (Section 5.1) or packing more coverage into the same ways. LeafPack's lead over the oracle therefore concentrates where active leaf demand exceeds what uncompressed protection can hold, as in the held-out mixes that include 4 GiB SysBench (7.7% and 5.7%), or where the freed capacity returns to application data, whether for a sensitive co-runner (10.0% on bc+omnetpp and 6.1% on XSBench+omnetpp) or for the program itself (8.4% and 10.2% on standalone DBx1000), while on the remaining held-out mixes it matches the oracle on average (+0.4% and +0.2%).

5.5 Replacement and Translation Baselines

SRRIP and Victima. To test a stronger replacement policy and compare fairly with Victima [33], Fig. 6b repeats our nine headline workloads using 3-bit SRRIP for application data. Because the SRRIP baseline improves performance by 1.8 to 21.9% over random replacement, LeafPack's gains over our default random-replacement baseline exceed its gains over the SRRIP baseline on these workloads, so we report both. Even on top of SRRIP, LeafPack delivers further gains, improving geometric-mean harmonic speedup over SRRIP by 18.7% on the small core and 17.7% on the server core across the nine headline workloads (and by 9.5% and 8.7% over the SRRIP baseline, or 4.5% and 4.4% over the best static uncompressed configuration on SRRIP, across all 18 development workloads).

Table 2: Held-out mixes and new benchmarks (*HS* / *vs. best static uncompressed*).

Workload	Small core (HS / vs. best static uncompressed)	Server core (HS / vs. best static uncompressed)
bfs+XSBench	1.151 / 1.007	1.163 / 1.012
bfs+bfs	1.286 / 1.024	1.250 / 1.027
XSBench+XSBench	1.317 / 1.021	1.261 / 0.985
pr+sssp	1.204 / 0.960	1.184 / 0.957
bc+bfs	1.350 / 1.049	1.303 / 1.045
bc+sssp	1.175 / 0.946	1.146 / 0.954
bfs+SysBench 4 GiB	1.038 / 1.017	1.038 / 1.015
bc+SysBench 4 GiB	1.416 / 1.213	1.386 / 1.193
pr+SysBench 4 GiB	1.335 / 1.174	1.273 / 1.109
sssp+bfs	1.067 / 0.988	1.049 / 0.997
pr+pr	1.196 / 1.002	1.202 / 1.009
bc+XSBench	1.352 / 1.045	1.383 / 1.088
sssp+XSBench	1.310 / 1.001	1.269 / 0.987
pr+XSBench	1.280 / 1.001	1.225 / 1.003
pr+sssp+XSBench+SysBench	1.280 / 1.062	1.218 / 1.034
bc+bfs+SysBench+omnetpp	1.069 / 1.028	1.044 / 1.012
pr+sssp+bfs+XSBench	1.235 / 0.986	1.207 / 0.988
pr+XSBench+SysBench+omnetpp	1.159 / 1.038	1.105 / 1.026
pr+bfs+SysBench+omnetpp	1.064 / 1.023	1.026 / 1.019
sssp+bfs+XSBench+omnetpp	1.099 / 1.022	1.061 / 0.982
Geometric mean (held-out)	1.214 / 1.029	1.185 / 1.021
DBx1000	1.154 / 1.084	1.196 / 1.102
mst	1.000 / 1.000	1.000 / 1.000
lbm	1.001 / 1.001	1.000 / 1.000
DBx1000+lbm	1.047 / 1.010	1.050 / 1.010
mst+lbm	1.006 / 0.999	1.005 / 1.000
DBx1000+mst	0.999 / 0.990	1.023 / 0.991
Geometric mean (new)	1.033 / 1.014	1.044 / 1.016

When we evaluate Victima on SRRIP using its best L2 way cap per workload, it achieves 0.90 to 1.03 of the SRRIP baseline on the headline workloads, whereas LeafPack on SRRIP outperforms Victima on every headline workload on both cores. Across thirteen 1 MiB workloads, Victima reaches 0.90 to 1.06 of the SRRIP baseline while LeafPack leads Victima by 13.7% and 14.9% in geometric mean. We also reduced our default page-walker count from 32 to 4 to increase the cost of page-table walks, which left this comparison unchanged. With a 2 MiB L2 cache, Victima breaks even on bc (1.004), while LeafPack still leads Victima by 20% and 21% across five workloads. This difference stems from walk cost in our Arm hierarchy, where 92% of L2 TLB misses hit a partial walk-cache entry in the L2 TLB and require only a single leaf read from the L2 cache. When we instead configure three 32-entry dedicated walk caches without L2-TLB partial translations to match Victima's original setup, Victima gains 6.1% on XSBench as expected, while LeafPack leads Victima by 4 to 6% on XSBench and 15 to 17% on bc.

Prioritization and equal-area alternatives. We model EWaH and xPTP [21, 53] as L2 replacement policies and compare each design against the baseline with the same data replacement policy, since xPTP relies on LRU recency order. While protecting page-table lines under EWaH speeds up standalone bc (1.27 HS), it hurts most co-runs with data-sensitive applications, dropping performance to 0.48 to 0.99 of the baseline under random replacement, 0.34 to 0.98 under SRRIP, and 0.26 to 0.99 under LRU. By turning off page-table priority when translation pressure is low, xPTP avoids most of that slowdown (0.83 at worst) but also gives up most of the translation speedup, reaching only 1.01 on 4 GiB SysBench and 1.17 on bc. Even when we compare against a per-workload oracle that picks the best prioritization scheme or the baseline for each run, LeafPack leads in geometric mean over 17 development workloads by 9.4% and 9.3% under random replacement, 6.4% and 6.1% under SRRIP (over EWaH or the baseline), and 5.6% and 4.9% under LRU (over EWaH, xPTP, or the baseline), falling below the LRU baseline only on the two graph and gcc co-runs (0.85 to 0.98). We also compare LeafPack against spending a comparable or larger area budget (at least 19.4 KiB or 0.050 mm^2, within 14 to 21% of our design's 0.058 to 0.063 mm^2) on conventional structures across bc, XSBench, SysBench, and bc+omnetpp. Doubling or quadrupling the L2 TLB (to 2,560 or 5,120 entries per core with one extra access cycle), adding a 16 KiB dedicated walk cache (3.1 to 3.4× our area), or adding an extra L2 way (1.2 to 1.3× our area) yields only 0.997 to 1.034 harmonic speedup, compared to 1.18 to 1.69 for LeafPack.

CoLT. Because CoLT [57] coalesces TLB entries only when consecutive virtual pages map strictly contiguous physical frames, which occurs infrequently in our interleaved multiprogrammed boots, CoLT alone achieves 0.995 to 1.058 harmonic speedup. However, combining LeafPack with CoLT preserves LeafPack's speedup within 0.7% (never more than 0.3% below it) across contiguous, interleaved, and fragmented mappings and never trails CoLT alone.

5.6 Scaling and Sensitivity

Core count and cache size. Scaling from one copy of bc on a 1 MiB L2 to eight copies sharing a 4 MiB L2 multiplies leaf demand by eight while only quadrupling cache capacity. At a 5-way ceiling,

compression cuts leaf L2 misses from 34.1 to 11.2 per 1,000 instructions (37.2 to 12.3 on the server core) and gains 15.2% and 14.9% over 5-way uncompressed protection, while our controller selects 15 compressed ways to reach 1.331 and 1.321 harmonic speedup. Conversely, enlarging the L2 from 1 to 4 MiB for a fixed workload allows more leaves to fit uncompressed (Fig. 9a), yet LeafPack never falls more than 1% below the best static uncompressed configuration on bc and XSBench at 2 and 4 MiB while still leading it by 2.8 to 4.4% on bc+omnetpp and by 43 to 58% on 4 GiB SysBench.

Physical contiguity and fragmentation. Because LeafPack relies on physical frame regularity, booting bc+bc with contiguous rather than interleaved page allocation raises 5-way compressed speedup from 1.264/1.230 to 1.363/1.317, whereas eight-core bc sees no gain from initial contiguity because each process repeatedly frees and reallocates two 64 MiB arrays per iteration, interleaving frames at run time. Under heavier fragmentation across three allocator histories generated by a synthetic fragmenter, our frozen controller stays within 0.7% of the best static setting on fragmented bc and 4 GiB SysBench. Fragmented XSBench exposes a limitation, as the controller selects its 3-way uncompressed candidate in two histories to reach 1.04 to 1.05 harmonic speedup on both cores, or 0.82 of the best static setting (1.28 at 10 ways, outside the candidate set), and reaches 0.955 and 0.926 of it in the third history. Because our evaluated controller charges every leaf-line miss to the leader set of the line's packed index even when an incompressible line lives in its normal set, 99.6% and 99.1% of the counted incompressible misses on fragmented XSBench land on compressed leader sets and push the controller toward the 3-way uncompressed policy. When we evaluate an ablation that instead charges each miss to the set that stores the line and delays compressed probing until four times the warm-up period (adding two counter bits), fragmented XSBench rises to 1.21 and 1.18 across all three histories while 21 other development and fragmented runs stay within 1% of our evaluated controller. We audited this rule across 108,448 counted misses, including directed tests that force policy flips, probe-mode transitions, and in-flight miss merges, and confirmed zero misattributed misses. Because we designed this change after our held-out evaluation, we report it here as an ablation and keep the evaluated controller unchanged.

Three-level hierarchy. Adding a private 256 KiB, 8-way L2 to each core while giving the shared L3 our default L2 geometry halves the leaf reads that reach the L3. Across the nine headline workloads and two gcc pairs in this hierarchy, LeafPack improves geometric-mean performance by 24.3% and 22.9% over the baseline and by 7.0% and 7.1% over the best static uncompressed configuration, closely matching their two-level results (7.5% and 7.6%). Because the private L2s also absorb most of gcc's data pressure, those two gcc pairs gain 0.4 to 1.5% here instead of losing about 4% as in the two-level system.

Huge pages, nesting, and translation hierarchy. When we sweep the L2 TLB from 640 to 5,120 entries and either disable the L2 prefetcher or replace it with a virtual-address stride and indirect prefetcher, LeafPack's speedup and its margin over uncompressed protection remain essentially unchanged across bc, XSBench, SysBench, and bc+omnetpp. When transparent huge pages [72] cover 98.9 to 99.9% of mapped memory, the allocated 4 KiB leaf footprint shrinks to 8 to 95 KiB and LeafPack stays within 0.6% of uncompressed protection (1.000 to 1.036 HS). Conversely, under two-dimensional nested paging with 4 KiB guest and host pages where LeafPack compresses leaf lines at both stages (using a fixed stage-2 identity map without VM exits), our controller improves harmonic speedup by 28 to 66% over the baseline and by 4 to 35% over the best static uncompressed configuration (Fig. 9b).

Latency, port contention, and seeds. LeafPack's gains are also insensitive to L2 datapath timing and simulation methodology. Eliminating our two- and three-cycle decode latency changes IPC by only about 0.2%, while adding an extra decode cycle or a two-cycle fill encoder changes harmonic speedup by at most 0.24%. Modeling a single-ported L2 array that serializes slot metadata and payload reads, converted-group probes, and dirty reclamation writebacks (without charging for clean victims or tag updates) changes harmonic speedup by only −0.7 to +0.4% across eight workloads. Locking the set during group transitions and capping transition writebacks at two write-buffer entries similarly changes harmonic speedup by at most 0.2% across 11 workloads on both cores. Evaluating workloads at fixed instruction counts rather than fixed cycle windows preserves all policy rankings across six tested workloads (changing LeafPack's margin over uncompressed protection by at most 3%), and varying the random seed across four leader-set hashes and replacement generators (eight for sssp+omnetpp and pr+gcc) changes harmonic speedup by only 0.1 to 0.8% on bc, SysBench, and Mix-3 and by 1.6% on pr+gcc. The sole exception is sssp+omnetpp, where one of the eight leader-set hashes causes the controller to select the 3-way uncompressed policy throughout the run and settle at 1.041 and 1.028 instead of 1.111 and 1.082.

5.7 Limitations and Policy Trade-offs

Co-runs with gcc. Across our development set, sssp+gcc and pr+gcc are the only workloads that trail the baseline by more than 0.5%. On both pairs, our controller selects the 5-way compressed policy and reaches 0.957 to 0.962 HS, trailing the 3-way uncompressed policy (0.988 to 1.004 HS). On pr+gcc, compressing leaf lines eliminates nearly all of pr's walk misses (1.37M to 0.01M), allowing pr to execute faster and issue 19% more data misses while gcc makes 20%

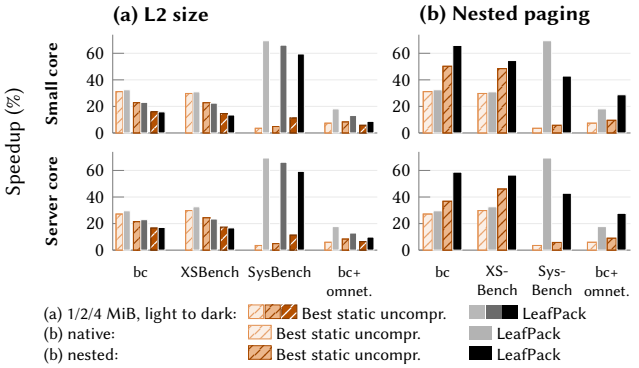


Figure 9: Sensitivity to L2 capacity and nested paging, showing harmonic speedup gain (%) over a same-size baseline across (a) 1, 2, and 4 MiB L2 caches (light to dark) and (b) native (light) versus nested (dark) paging.

less progress (1.16M versus 1.45M instruction accesses). This slowdown stems from L2 data capacity contention rather than memory bandwidth, as dedicating even three uncompressed ways to page tables reduces gcc to 0.91 to 0.96 of its progress under the baseline, and expanding to five compressed ways raises gcc's L2 misses per instruction by another 15 to 25% while DRAM read latency and bus utilization change by at most 6% and 3.3 percentage points. Our controller picks the 5-way policy on these two pairs because its shadow tags observe the access stream generated under the compressed leader sets rather than a physical uncompressed cache, which overestimates the 3-way policy's data misses by 6 to 8% and reverses a true difference of 0 to 5%. On every other development workload and core, our controller selects the uncompressed policy whenever it lies within 1% of the best.

Choice of candidate policies. Our two compressed candidates serve two distinct operating regimes. The 15-way ceiling excels on large active leaf sets, reaching 1.71 versus 1.11 static harmonic speedup on 4 GiB SysBench and 1.38 versus 1.26 on bc+bc while occupying 66 to 94% of the L2. Conversely, the 5-way ceiling caps page-table occupancy at 31% of the L2 to protect data-sensitive co-runners, reaching 1.18 versus 1.02 on bc+omnetpp and 0.97 versus 0.46 on sssp+gcc. While a 4-way partial group improves Mix-2 and the gcc pairs slightly, we restrict our controller to three candidate policies because evaluating a fourth candidate would require dedicating another 32 physical leader sets and adding more pairwise comparison counters.

Scope and future work. LeafPack targets the page walks that remain after huge pages, where memory stays PTE-mapped yet retains short runs of consecutive physical frames. Because each compressible leaf line needs only eight consecutive frames (32 KiB) rather than an aligned 2 MiB block, this regime persists when operators disable transparent huge pages for tail latency [61], when the kernel splits a huge-page mapping without breaking up its backing allocation, and under modern multi-size huge pages that remain PTE-mapped [74, 75]. Our native Arm experiments with 4 KiB pages evaluate this regime directly. Two-dimensional page walks expand the active leaf working set even further, and our nested-paging proof of concept in Section 5.6 improves harmonic speedup over the baseline by 28 to 66% under a fixed stage-2 identity map without VM exits. We leave modern-kernel huge-page policies, full virtualization, and a held-out evaluation of our fill-set attribution ablation to future work.

6 Related Work

Page-table and translation caching. Page-walk caches keep upper-level descriptors close to the MMU [10, 14] while leaving leaf descriptors to the data caches. Like EWaH and xPTP [21, 53] (Section 2), LeafPack retains leaf lines in the shared cache, but compresses regular leaf lines into shared slots under a bounded way ceiling to preserve data capacity. By contrast, rather than caching page-table lines, Victima [33] stores completed translations in L2 blocks, DUCATI [28] lets a GPU last-level cache hold TLB entries alongside data and backs them with a TLB in DRAM, POM-TLB [63] places a large translation table in DRAM whose entries are cached in the data hierarchy, and near-memory walkers execute walks inside the memory controller [58]. By compressing and caching the architectural page-table lines themselves instead of maintaining a secondary store of completed translations, LeafPack leaves the walk algorithm and TLB shootdowns unchanged.

TLB reach, contiguity, and prefetching. Coalescing and clustered TLBs combine contiguous or nearby page translations into single TLB entries [52, 56, 57, 71], and operating-system and memory-allocator policies increase physical contiguity to expose more coalescing opportunities [5, 27, 39, 45, 47–49, 83]. Whenever a workload's active footprint exceeds the reach of the coalesced TLBs, however, TLB misses still trigger hardware page-table walks through the cache hierarchy. Translation prefetchers that predict upcoming translations [16, 18, 31, 44, 77, 78] and speculative address translation designs [35] similarly fetch their page table descriptors through the data caches. Because Trail [34] also embeds per page prefetch deltas in unused PTE bits, it can prevent LeafPack from compressing a leaf line whenever those bits differ across its eight PTEs. We exploit this spatial regularity inside the shared cache so that both demand and prefetched walks hit on chip, composing cleanly with CoLT (Section 5.5) and benefiting from contiguous operating-system page allocation (Section 5.6).

Alternative page-table organizations. Elastic cuckoo page tables replace dependent radix tree lookups with parallel hash lookups [68], Compendia selectively merges radix-tree levels [2], and Utopia and direct segment-like translation schemes constrain virtual-to-physical placement so large regions can be translated in a single step [3, 11, 32, 36, 82]. Virtualized architectures similarly accelerate two-dimensional guest-to-host walks through nested walk caches or flattened intermediate tables [1, 13, 24, 25]. Whereas those designs modify both the hardware page-table walkers and the operating system's memory manager, LeafPack keeps the standard radix page-table layout and operating-system interface unchanged.

Cache compression and adaptive management. General-purpose compression algorithms such as BDI, FPC, C-Pack, FVC, SC^2, and HyComp encode arbitrary data blocks [4, 6, 7, 22, 55, 80], while compressed-cache layouts such as DCC and SCC track neighboring compressed blocks under shared superblock tags [65, 67], and YACC [66] adds per-block coherence state. Rather than expanding page table residency in the caches, TMCC [50] and DyLeCT [51] accelerate the physical to DRAM address lookup required by hardware compressed main memory, with TMCC storing compression metadata inside page table blocks. Building on superblock tag sharing, we target shared-cache page-table residency by managing compressed leaf lines as a dedicated cache class under an adaptive way ceiling and tailoring our packed (6.4×) and dense (12.8×) formats to regular page-table descriptors within five-way groups. Adaptive compressed caches decide at runtime whether compression pays off [4], while CAMP uses compressed block size to guide replacement [54]. Our runtime controller builds on set dueling [29, 59] and shared-cache partitioning [20, 60, 64, 70, 79] by adding shadow-tag evaluation and per-core miss weighting.

7 Conclusion

We present LeafPack, a shared-cache design that packs regular leaf page-table lines into compact groups while preserving architectural page-table formats, per-line coherence, and uncompressed

fallback for irregular lines. By combining our cross-line descriptor format with an online controller that adapts the page-table way ceiling, LeafPack either retains larger leaf working sets in the cache or delivers the same translation coverage in fewer ways so co-running applications keep more room for their data. Across our nine translation-heavy workloads on small and server cores, LeafPack improves geometric-mean harmonic speedup over the baseline by 28.7% and 28.4% and speeds up all twenty held-out mixes.

Acknowledgments. We used generative AI tools (OpenAI ChatGPT and Codex, and Anthropic Claude) to assist with drafting and revising text, code, experimental scripts, tables, and figures. We reviewed all generated material, verified all sources and results, and take full responsibility for the work.

References

[1] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 476–487. doi:10.1109/ISCA.2012.6237041

[2] Sam Ainsworth and Timothy M. Jones. 2021. Compendia: reducing virtual-memory costs via selective densification. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 52–65. doi:10.1145/3459898.3463902

[3] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 457–468. doi:10.1145/3079856.3080209

[4] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (München, Germany) *(ISCA '04)*. IEEE Computer Society, USA, 212.

[5] 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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 515–528. doi:10.1109/ISCA45697.2020.00050

[6] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: a hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/2830772.2830823

[7] Angelos Arelakis and Per Stenstrom. 2014. SC2: a statistical compression cache scheme. In *Proceedings of the 41st Annual International Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 145–156.

[8] Arm Limited. 2026. Arm Architecture Reference Manual for A-profile Architecture. https://support.arm.com/documentation/ddi0487/md. ARM DDI 0487, Issue M.d.

[9] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. doi:10.1145/3085572

[10] 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)*. Association for Computing Machinery, New York, NY, USA, 48–59. doi:10.1145/1815961.1815970

[11] 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)*. Association for Computing Machinery, New York, NY, USA, 237–248. doi:10.1145/2485922.2485943

[12] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[13] 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)*. Association for Computing Machinery, New York, NY, USA, 26–35. doi:10.1145/1346281.1346286

[14] 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)*. Association for Computing Machinery, New York, NY, USA, 383–394. doi:10.1145/2540708.2540741

[15] Abhishek Bhattacharjee. 2017. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro* 37, 5 (Sept. 2017), 6–10. doi:10.1109/MM.2017.3711640

[16] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 63–76. doi:10.1145/3037697.3037705

[17] Abhishek Bhattacharjee and Daniel Lustig. 2017. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers. doi:10.2200/S00795ED1V01Y201708CAC042

[18] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 359–370. doi:10.1145/1736020.1736060

[19] 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. doi:10.1145/2024716.2024718

[20] Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing* (Seattle, Washington) *(ICS '07)*. Association for Computing Machinery, New York, NY, USA, 242–252. doi:10.1145/1274971.1275005

[21] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A. Jiménez, and Marc Casas. 2025. Instruction-Aware Cooperative TLB and Cache Replacement Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 619–636. doi:10.1145/3669940.3707247

[22] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: a high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8 (Aug. 2010), 1196–1208. doi:10.1109/TVLSI.2009.2020989

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–189. doi:10.1109/MICRO.2014.37

[25] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 707–718. doi:10.1109/ISCA.2016.67

[26] gem5 developers. 2026. gem5::CompressedTags Class Reference. https://doxygen.gem5.org/develop/classgem5_1_1CompressedTags.html. gem5 documentation, accessed November 15, 2026.

[27] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[28] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Trans. Archit. Code Optim.* 16, 1, Article 6 (March 2019), 24 pages. doi:10.1145/3309710

[29] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 208–219.

[30] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 60–71. doi:10.1145/1815961.1815971

[31] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska) *(ISCA '02)*. IEEE Computer Society, USA, 195–206.

[32] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, F. Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu. 2023. Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. ACM, 1196–1212. doi:10.1145/3613424.3623789

[33] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. 2023. Victima: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. ACM, 1178–1195. doi:10.1145/3613424.3614276

[34] Konstantinos Kanellopoulos, Konstantinos Sgouras, Harsh Songara, Rahul Bera, and Onur Mutlu. 2026. Trail: Scalable and Low-Cost Temporal TLB Prefetching via Page-Table-Embedded Deltas. arXiv:2610.08483 [cs.AR] https://arxiv.org/abs/2610.08483

[35] Konstantinos Kanellopoulos, Konstantinos Sgouras, Harsh Songara, Andreas Kosmas Kakolyris, Vlad-Petru Nitu, Spiros Galanopoulos, Rahul Bera, Konstantina Koliogeorgi, Rakesh Kumar, and Onur Mutlu. 2026. Revelator: Rapid Data

Fetching Via System-Software-Guided Hash-Based Speculative Address Translation . In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 2061–2080. doi:10.1109/ISCA66397.2026.00147

[36] 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)*. Association for Computing Machinery, New York, NY, USA, 66–78. doi:10.1145/2749469.2749471

[37] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, USA, 111–122.

[38] Alexey Kopytov. 2020. SysBench: A Scriptable Database and System Performance Benchmark. https://github.com/akopytov/sysbench. Version 1.0.20.

[39] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[40] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] https://arxiv.org/abs/2007.03152

[41] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 164–171. doi:10.1109/ISPASS.2001.990695

[42] Mahesh Madhav, Allen Lee, Andres Mejia, Branden Moore, Charan Soppadandi, Chris Cambly, Christoph Müllner, Daniel Bowers, David Reiner, Denis Bakhvalov, Di Zhao, Duane Voth, Feng Xue, Frédérique Silber-Chaussumier, James Bucek, James Southern, Jiangning Liu, Jim Himer, John Henning, Kevin Smith, Kristen Yang, Kunal Kashyap, Mason Guy, Mat Colgrove, Michael Berg, Prasad Battini, Prasad Joshi, Rohit Prasad, Shay Bhattacharya, Sriyash Caculo, Stefan Reimbold, Sundar Iyengar, Van Smith, and Zarko Todorovski. 2026. SPEC CPU: The Next Generation. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 671–687. doi:10.1109/isca66397.2026.00059

[43] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: context switch aware large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 449–462. doi:10.1145/3123939.3124549

[44] 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)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. doi:10.1145/3352460.3358294

[45] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. 2021. PTEMagnet: fine-grained physical memory reservation for faster page walks in public clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 211–223. doi:10.1145/3445814.3446704

[46] David S. Miller. 2026. Cache and TLB Flushing Under Linux. https://docs.kernel.org/core-api/cachetlb.html. Linux kernel documentation, accessed November 15, 2026.

[47] Juan Navarro, Sitaram Iyer, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi-02/practical-transparent-operating-system-support-superpages

[48] 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)*. Association for Computing Machinery, New York, NY, USA, 347–360. doi:10.1145/3297858.3304064

[49] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. *SIGPLAN Not.* 53, 2 (March 2018), 679–692. doi:10.1145/3296957.3173203

[50] Gagandeep Panwar, Muhammad Laghari, David Bears, Yuqing Liu, Chandler Jearls, Esha Choukse, Kirk W. Cameron, Ali R. Butt, and Xun Jian. 2022. Translation-optimized Memory Compression for Capacity. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 992–1011. doi:10.1109/MICRO56248.2022.00073

[51] Gagandeep Panwar, Muhammad Laghari, Esha Choukse, and Xun Jian. 2024. DyLeCT: Achieving Huge-Page-Like Translation Performance for Hardware-Compressed Memory. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 1129–1143. doi:10.1109/ISCA59077.2024.00085

[52] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 444–456. doi:10.1145/3079856.3080217

[53] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 128–141. doi:10.1145/3503222.3507718

[54] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 51–63. doi:10.1109/HPCA.2015.7056021

[55] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 377–388. doi:10.1145/2370816.2370870

[56] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. doi:10.1109/HPCA.2014.6835964

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

[58] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. 2017. Near-Memory Address Translation. arXiv:1612.00445 [cs.AR] https://arxiv.org/abs/1612.00445

[59] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 381–391. doi:10.1145/1250662.1250709

[60] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 423–432. doi:10.1109/MICRO.2006.49

[61] Redis. 2026. Diagnosing Latency Issues. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/. Accessed November 16, 2026.

[62] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 233–263. doi:10.1145/201059.201065

[63] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 469–480. doi:10.1145/3079856.3080210

[64] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 57–68. doi:10.1145/2000064.2000073

[65] Somayeh Sardashti, André Seznec, and David A. Wood. 2014. Skewed Compressed Caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, USA, 331–342. doi:10.1109/MICRO.2014.41

[66] Somayeh Sardashti, Andre Seznec, and David A. Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. *ACM Trans. Archit. Code Optim.* 13, 3, Article 27 (Sept. 2016), 25 pages. doi:10.1145/2976740

[67] Somayeh Sardashti and David A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–73.

[68] 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)*. Association for Computing Machinery, New York, NY, USA, 1093–1108. doi:10.1145/3373376.3378493

[69] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[70] G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28, 1 (April 2004), 7–26. doi:10.1023/B:SUPE.0000014800.27383.8f

[71] 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)*. Association for Computing Machinery, New York, NY, USA, 171–182. doi:10.1145/195473.195531

[72] The Linux Kernel Documentation. 2018. Transparent Hugepage Support. https://www.kernel.org/doc/html/v4.18/admin-guide/mm/transhuge.html. Linux 4.18.

[73] The Linux Kernel Documentation. 2026. Page Tables. https://docs.kernel.org/mm/page_tables.html. Accessed November 15, 2026.

[74] The Linux Kernel Documentation. 2026. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html. Admin guide, accessed November 16, 2026.

[75] The Linux Kernel Documentation. 2026. Transparent Hugepage Support. https://docs.kernel.org/mm/transhuge.html. Memory management documentation, accessed November 16, 2026.

[76] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench – The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *International Conference on Physics of Reactors (PHYSOR)*.

[77] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1138–1153. doi:10.1145/3466752.3480049

[78] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 85–98. doi:10.1109/ISCA52012.2021.00016

[79] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 174–183. doi:10.1145/1555754.1555778

[80] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*. IEEE Computer Society, 258–265. doi:10.1109/MICRO.2000.898076

[81] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: an evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. doi:10.14778/2735508.2735511

[82] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. 2024. Direct Memory Translation for Virtualized Clouds. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 287–304. doi:10.1145/3620665.3640358

[83] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi