Matryoshka B+ Tree: Insert/Delete Performance Report

Comparative Benchmark Results

2026-02-20T19:48:00

Parameter	**Value**
CPU	13th Gen Intel(R) Core(TM) i7-1370P
L1d Cache	48 KB
L2 Cache	2 MB
L3 Cache	24 MB
Kernel	6.17.10-300.fc43.x86_64
Page Size	4096 B

Generated by `bench_compare` — matryoshka project

Contents

1 Introduction

This report evaluates the **matryoshka** B+ tree — a B+ tree whose page-sized leaf nodes contain nested B+ sub-trees of cache-line-sized (64 B) sub-nodes, with SIMD-accelerated search at every level — against several tree and ordered-map libraries on *insert-heavy* and *delete-heavy* workloads. Goals:

1. Quantify the modification throughput gap across dataset sizes (65,536 to 16,777,216 keys).
2. Identify micro-architectural bottlenecks (cache misses, TLB pressure, branch misprediction) that explain the differences.

All measurements use `clock_gettime(CLOCK_MONOTONIC)`. Results are reported as Mop/s and ns/op.

2 Library Descriptions

Table 1: Libraries under test.

Name	Label	Description
`matryoshka`	Matryoshka B+ tree	B+ tree with nested CL sub-tree leaves (up to 855 keys/page), SIMD
`std_set`	std::set (RB tree)	Red-black tree (libstdc++), pointer-chasing, 40–48 B/node
`tlx_btree`	TLX btree_set	Cache-conscious B+ tree, sorted-array leaves ($B \approx 128$)
`libart`	libart (ART)	Adaptive Radix Tree, 4-byte keys, no predecessor search
`abseil_btree`	Abseil btree_set	Google B-tree, sorted-array leaves ($B \approx 256$)

3 Workload Descriptions

Table 2: Benchmark workloads.

Workload	Description
`seq_insert`	Insert N keys in ascending order. Exercises append paths.
`rand_insert`	Insert N unique keys in random order. Stresses leaf splits.
`ycsb_a`	95% insert / 5% search. Write-dominated OLTP model.
`rand_delete`	Bulk-load N sorted keys, delete all in random order.
`mixed`	Bulk-load N keys, then N alternating insert/delete ops.
`ycsb_b`	Bulk-load N keys, then 50% delete / 50% search.
`search_after_churn`	Bulk-load N keys, $N/2$ mixed churn (untimed), then 5,000,000 random predecessor searches.

4 Results: Insert-Heavy Workloads

4.1 Sequential Insert

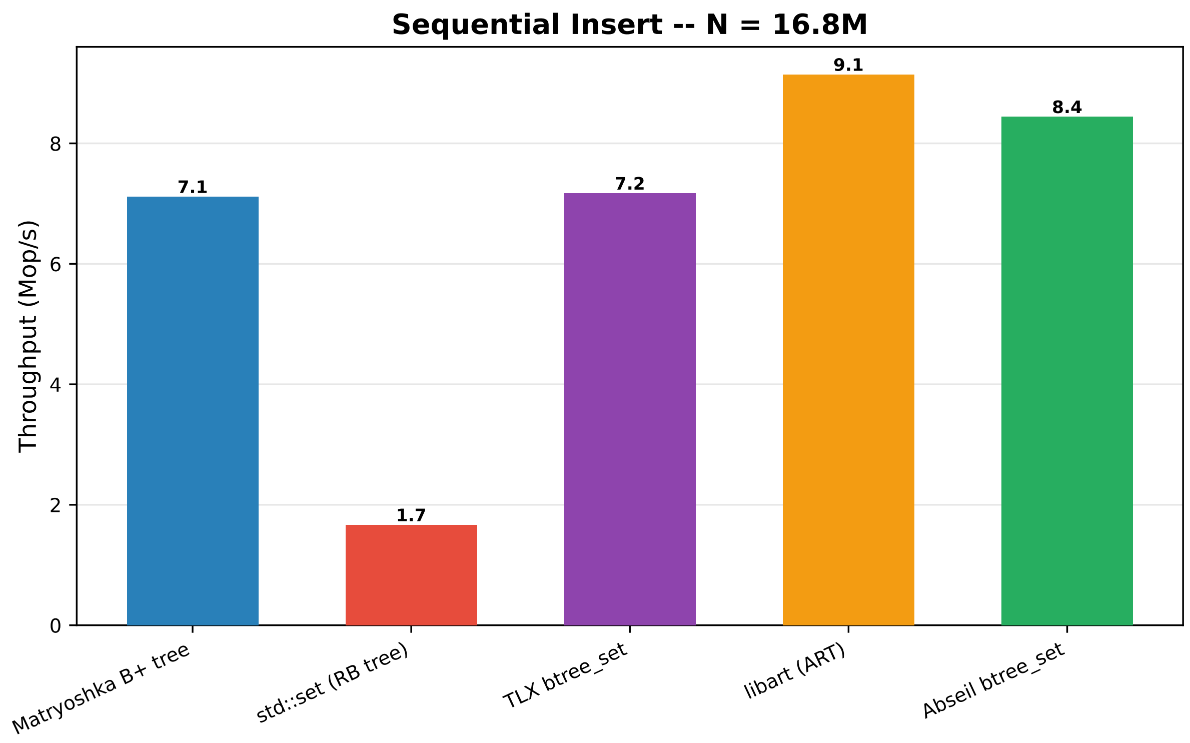

Figure 1: Sequential insert throughput (Mop/s).

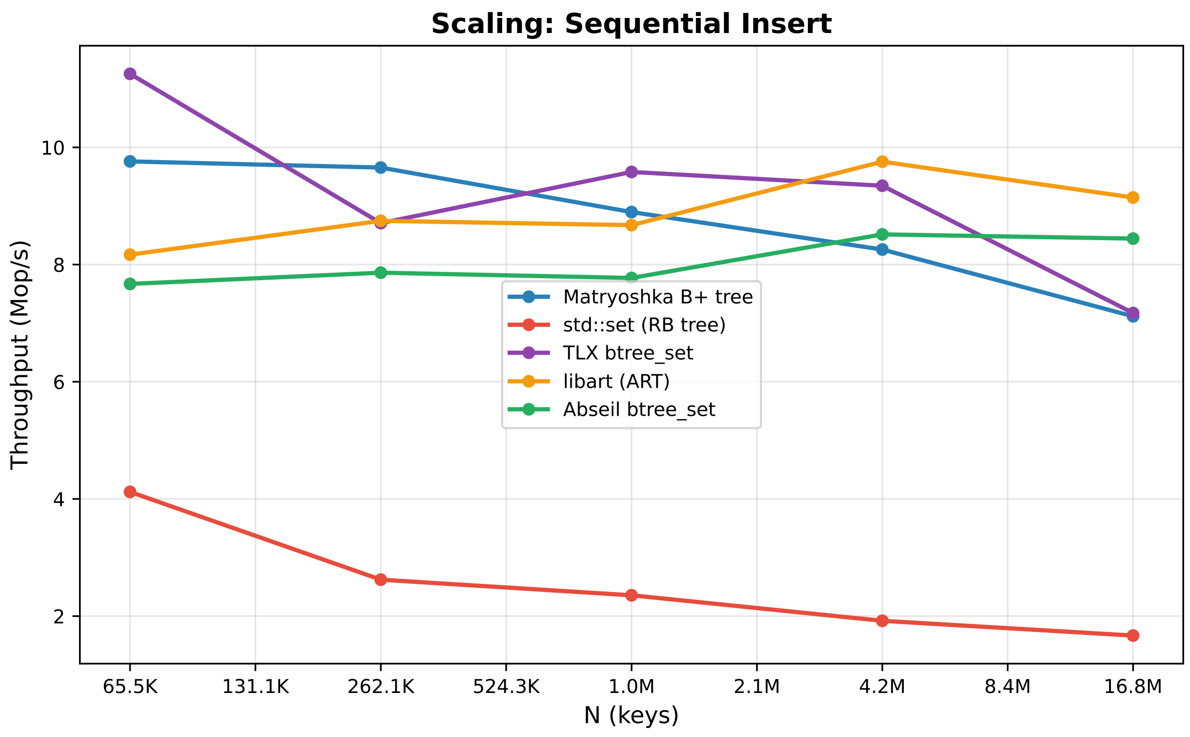

Figure 2: Sequential insert scaling.

4.2 Random Insert

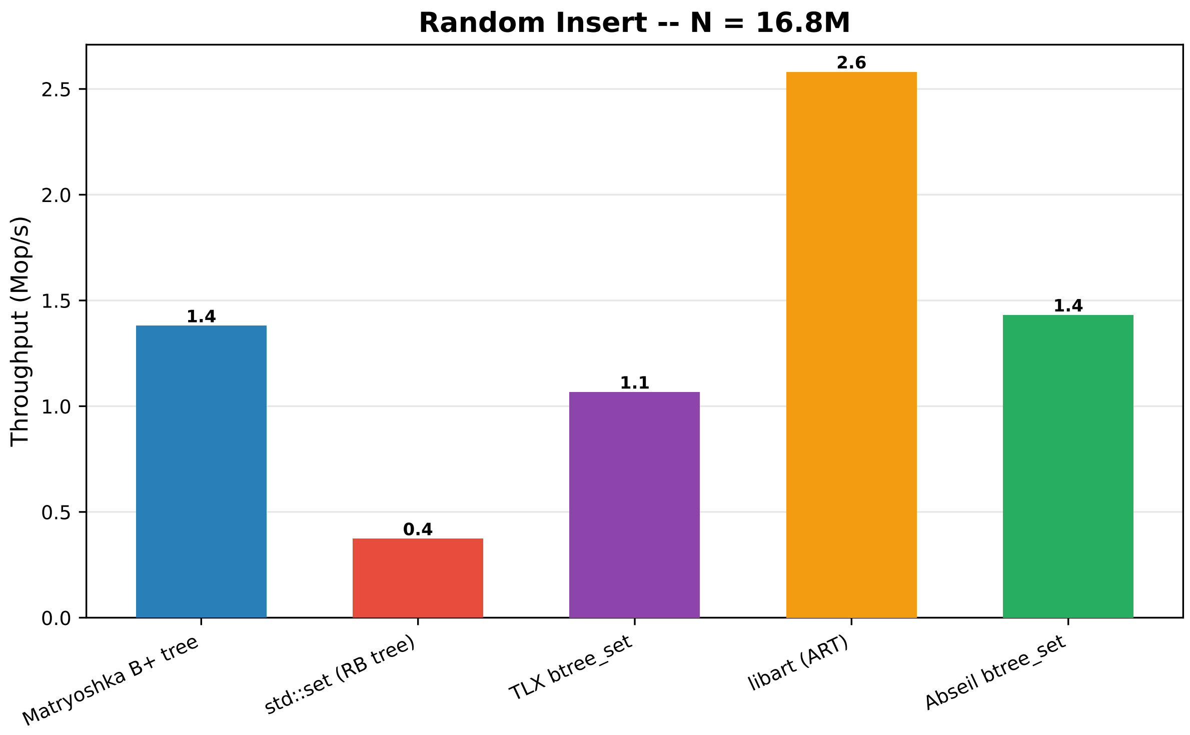

Figure 3: Random insert throughput (Mop/s).

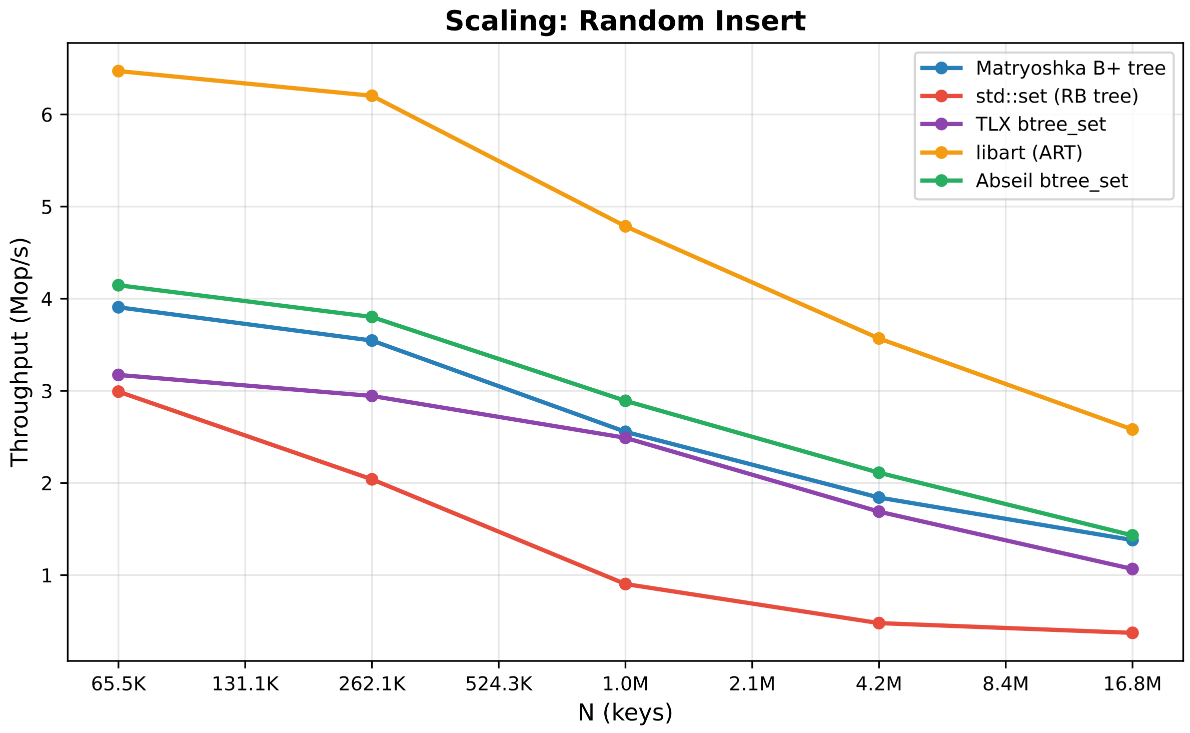

Figure 4: Random insert scaling.

4.3 YCSB-A (95% Insert / 5% Search)

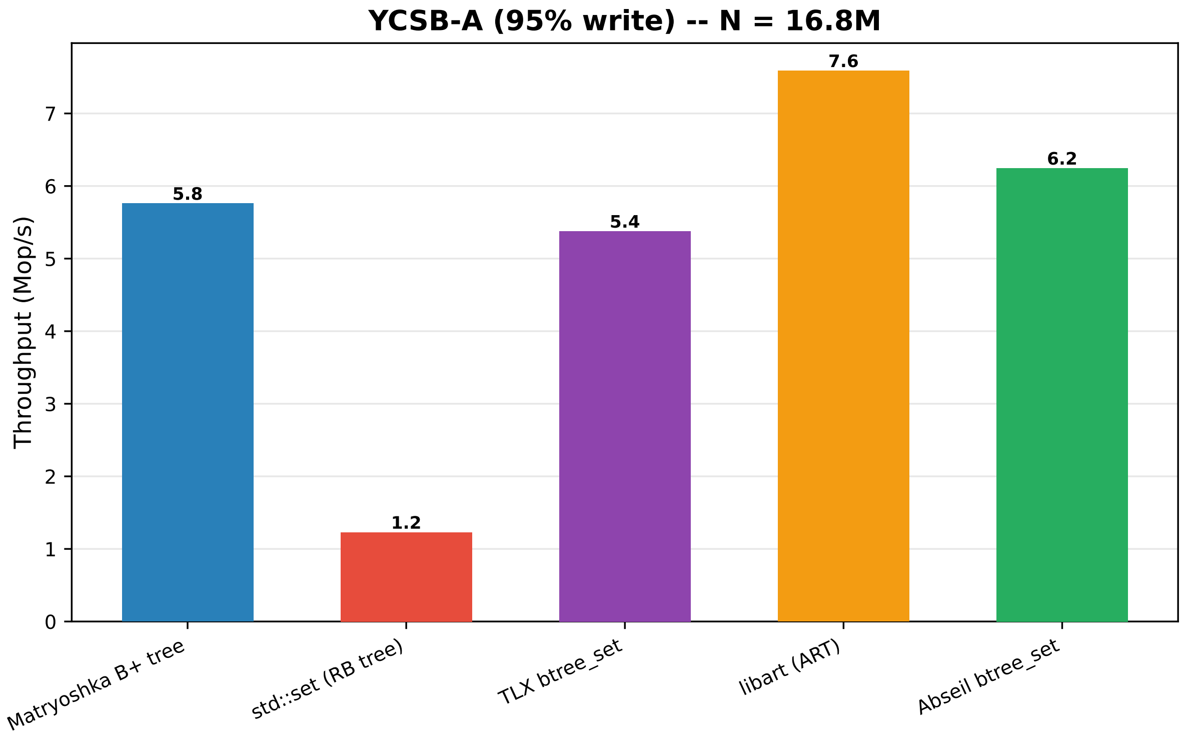

Figure 5: YCSB-A throughput (Mop/s).

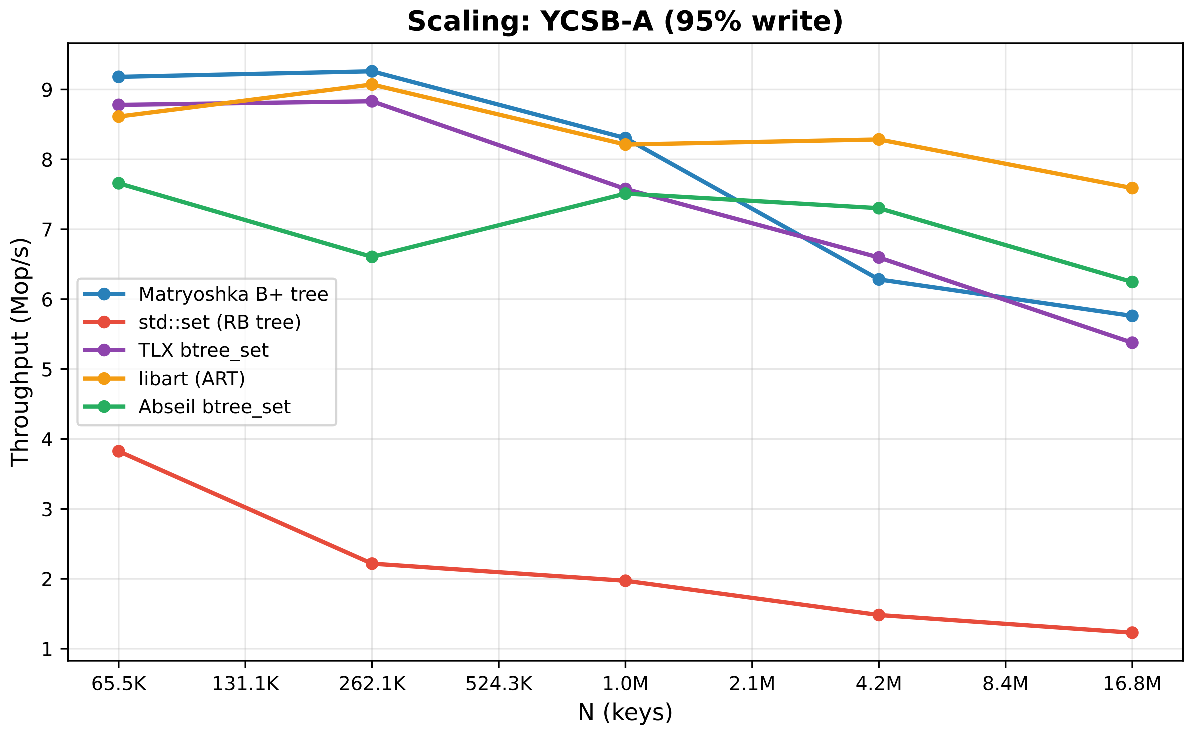

Figure 6: YCSB-A scaling.

5 Results: Delete-Heavy Workloads

5.1 Random Delete

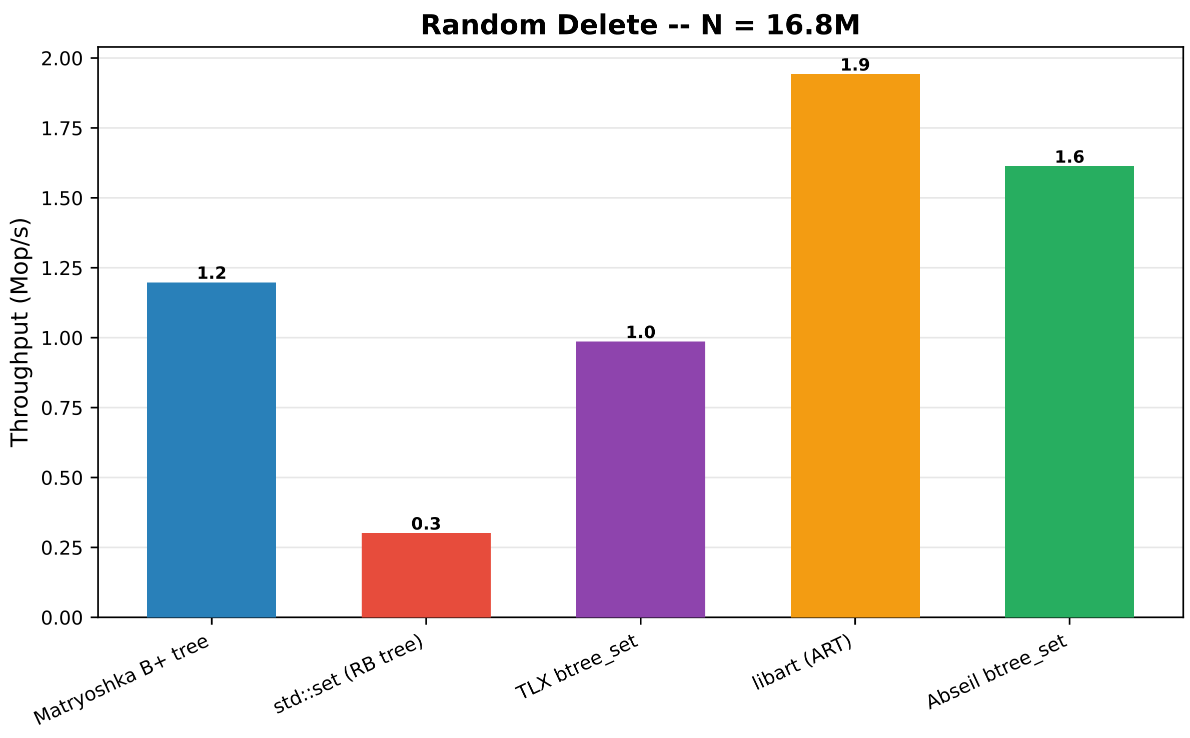

Figure 7: Random delete throughput (Mop/s).

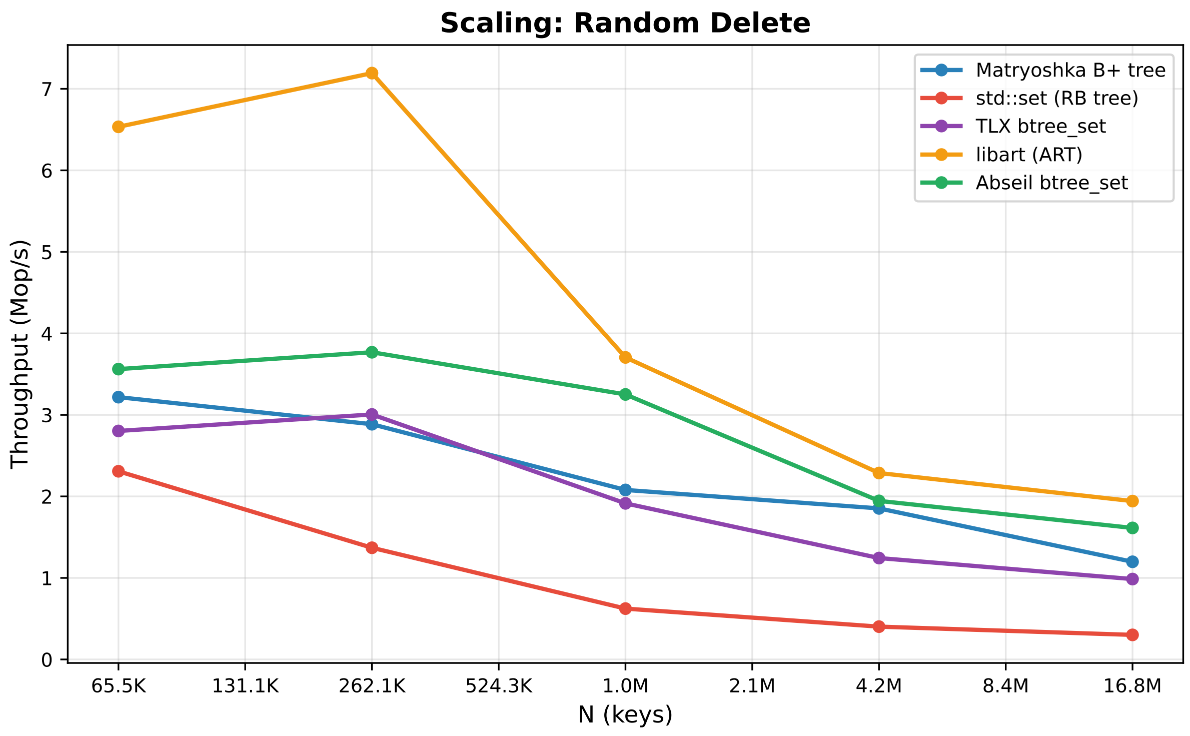

Figure 8: Random delete scaling.

5.2 Mixed Insert/Delete

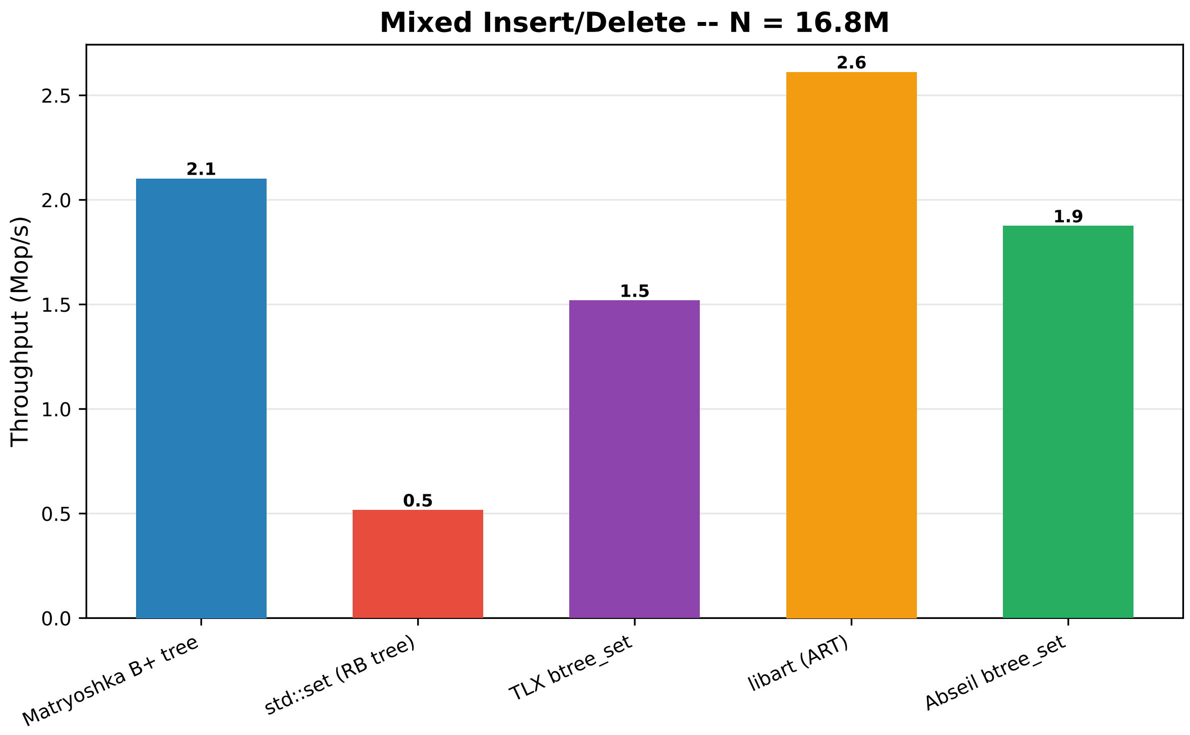

Figure 9: Mixed insert/delete throughput (Mop/s).

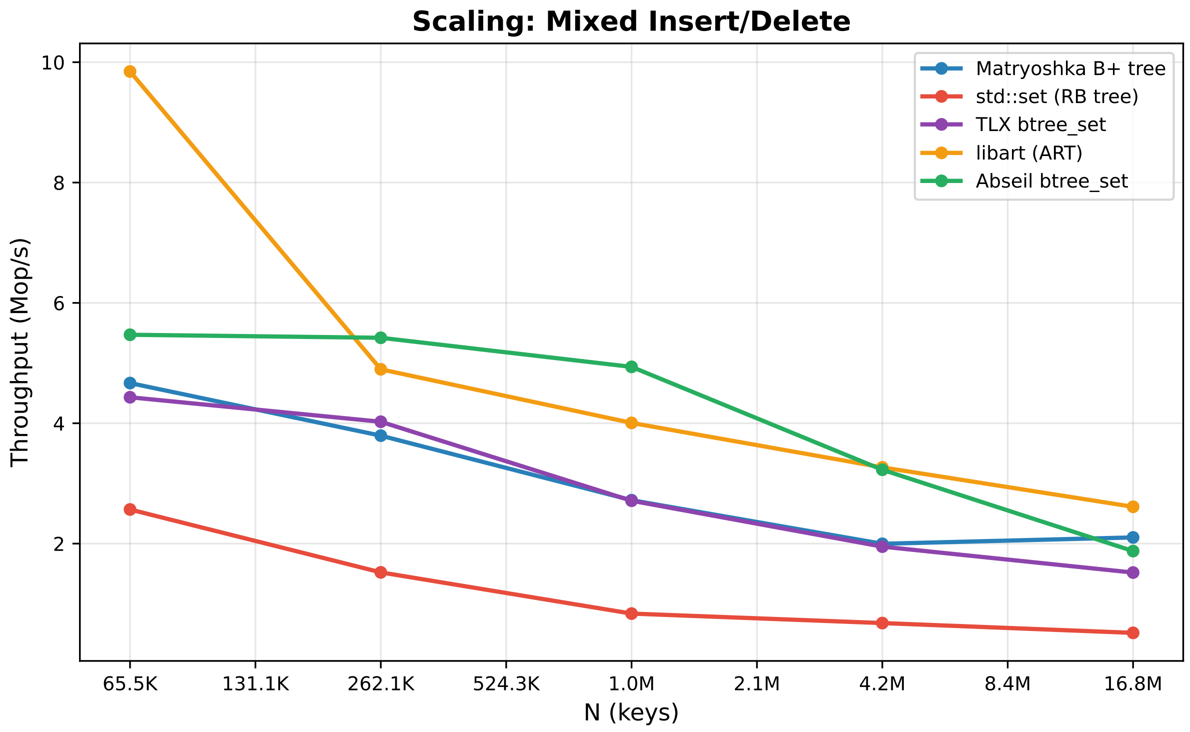

Figure 10: Mixed insert/delete scaling.

5.3 YCSB-B (50% Delete / 50% Search)

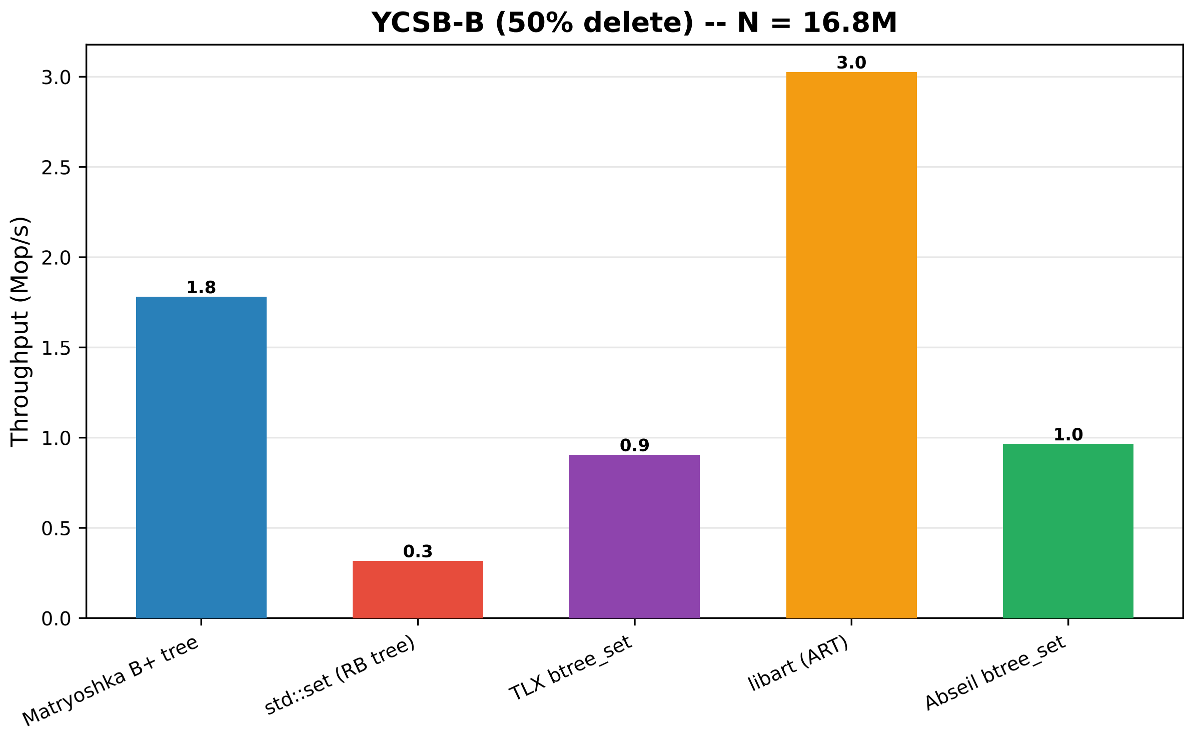

Figure 11: YCSB-B throughput (Mop/s).

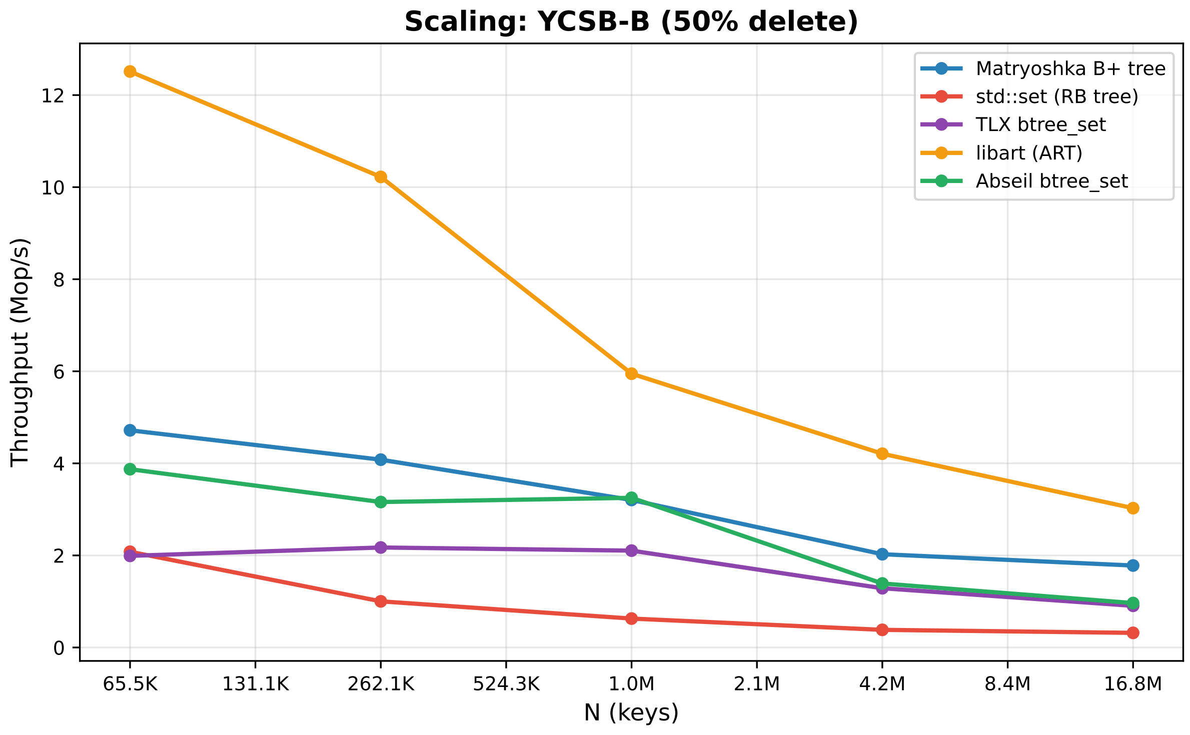

Figure 12: YCSB-B scaling.

6 Results: Search After Churn

The `search_after_churn` workload measures pure search throughput on a tree that has undergone insert/delete churn, isolating search performance from modification cost.

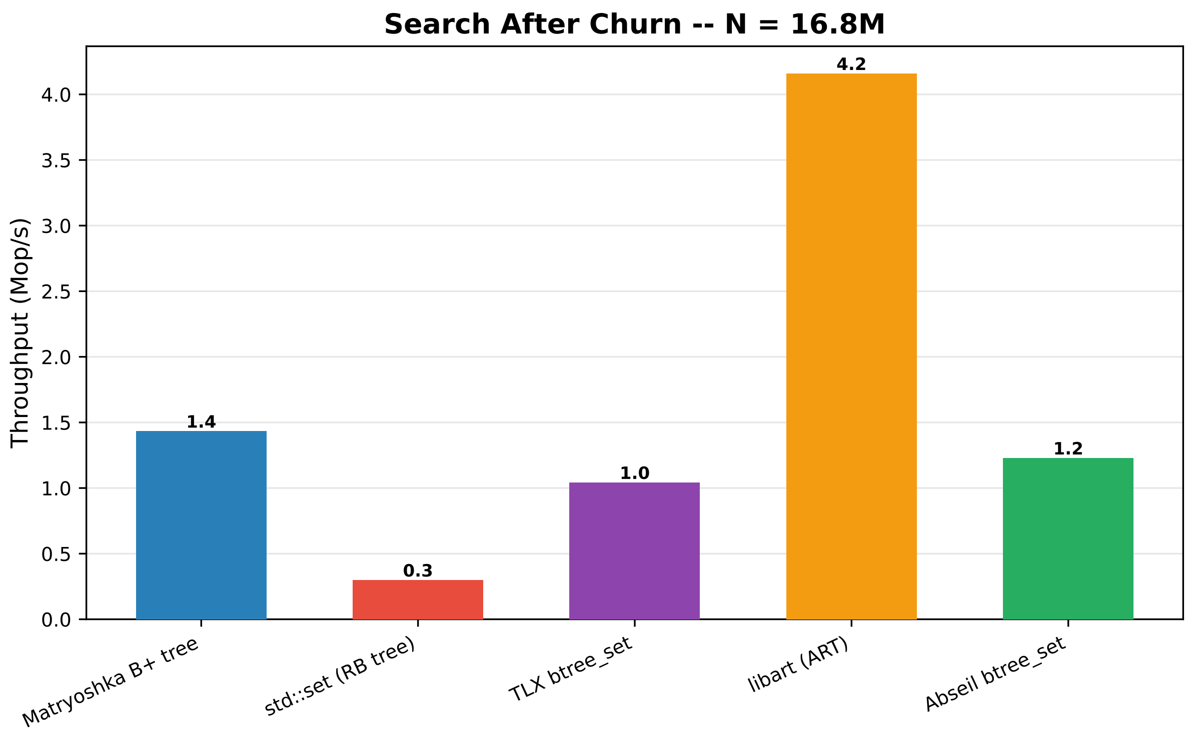

Figure 13: Search throughput after churn (Mop/s).

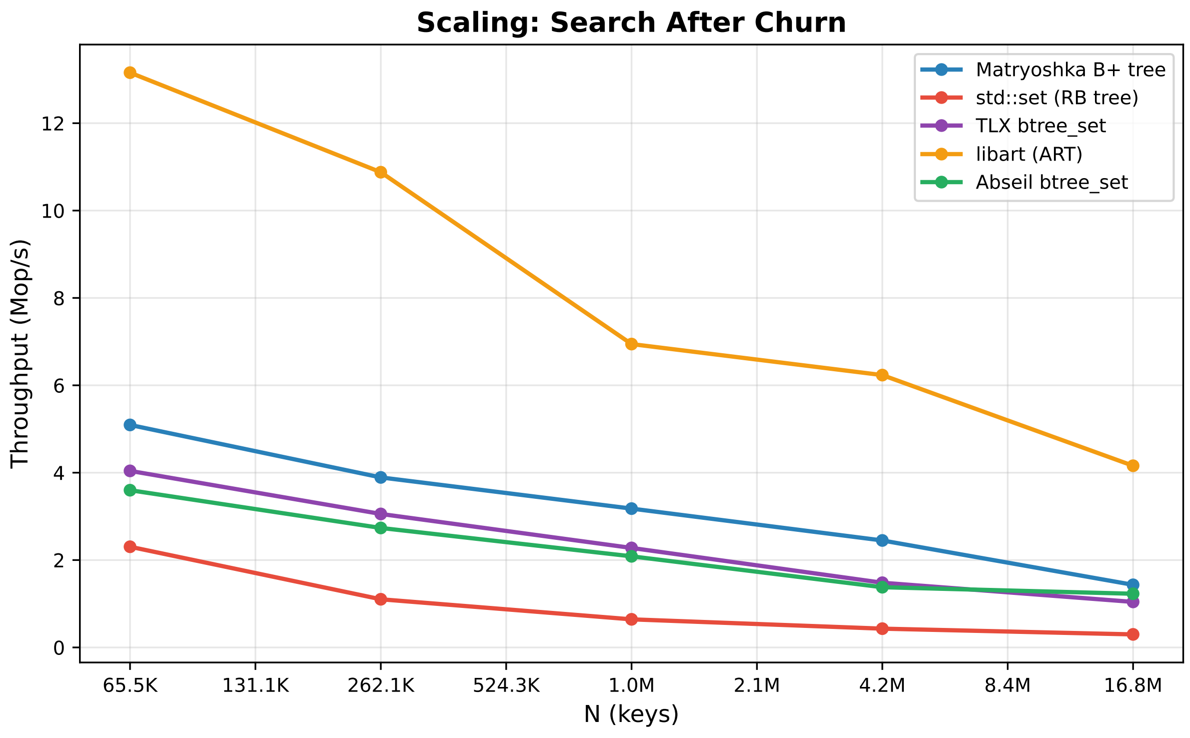

Figure 14: Search-after-churn scaling.

7 Hardware Counter Analysis

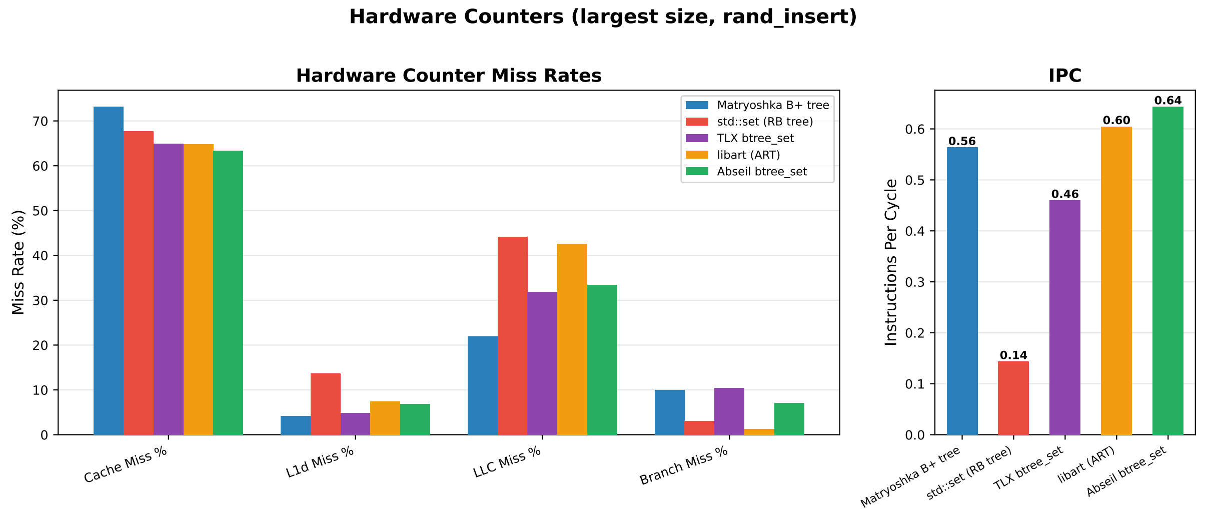

Figure 15: Hardware counters: dTLB miss rate, LLC miss rate, IPC, branch misprediction rate.

7.1 dTLB Miss Rate

Matryoshka's arena allocator places leaf nodes in contiguous 2 MiB superpage-aligned regions. At $N = 16{,}777{,}216$, matryoshka's dTLB miss rate is 2.8 per 1,000 ops, versus 66.9 for `std::set`. Red-black tree pointer chasing touches a new TLB entry per level; matryoshka confines each leaf search to a single 4 KiB page.

7.2 LLC Miss Rate

Matryoshka packs up to 855 keys per 4 KiB page using a nested B+ sub-tree of cache-line sub-nodes. `std::set` requires one 40–48 B heap node per key. At $N = 16{,}777{,}216$: 219.7 LLC misses/1,000 ops (matryoshka) vs. 441.8 (`std::set`).

7.3 IPC

SIMD search at every level (CL leaves, CL internals, outer internal nodes) achieves IPC of 0.56 via pipelined `_mm_cmpgt_epi32`/`_mm_movemask_ps` without data-dependent branches. Insert and delete paths also benefit from SIMD navigation through the CL sub-tree to locate the target sub-node.

7.4 Branch Misprediction

SIMD mask arithmetic replaces conditional branches during search, yielding near-zero misprediction. Modification operations navigate the CL sub-tree using the same SIMD search, with only the final CL leaf insert/delete using scalar `memmove` on ≤ 14 keys.

8 Profiling: Hot Functions

Table 3: Top functions (`perf record`, rand_insert, N=1,048,576).

% Overhead	Function	Source
79.8%	`page_find_leaf`	`bench_compare`
54.9%	`mt_page_insert`	`bench_compare`
38.5%	`mt_inode_search`	`bench_compare`
14.7%	`matryoshka_insert`	`bench_compare`
3.3%	`void workload_rand_insert<WrapperMatryoshka>(unsigned long)`	`bench_compare`
1.1%	`extract_subtree`	`bench_compare`

The hot functions are the page-level sub-tree operations: `mt_page_insert` and `mt_page_delete` navigate the CL sub-tree via SIMD search, then perform a scalar insert or delete within a single 64 B cache-line sub-node. CL sub-node splits and merges occur only on overflow or underflow.

9 Cache-Miss Attribution Analysis

This section attributes cache misses to individual functions using `perf record -e cache-misses`. The data reveals *where* in the code the processor stalls waiting for memory, complementing the aggregate hardware counters in Section 7.

Table 4: Top functions by cache-miss contribution (matryoshka, rand_insert, N=4,194,304).

% Cache Misses	Function
74.7%	`page_find_leaf`
52.2%	`mt_page_insert`
35.6%	`void workload_rand_insert<WrapperMatryoshka>(unsigned long)`
21.1%	`mt_inode_search`
5.8%	`matryoshka_insert`
1.1%	`extract_subtree`

9.1 Root Cause: CL Sub-Node Cache-Line Jumps

The dominant cache-miss source is the CL sub-tree navigation within each 4 KiB page leaf. When `page_find_leaf` descends the CL sub-tree, each level accesses a different 64 B slot within the page. These slots are not adjacent — slot indices are assigned by a bitmap allocator, so the root CL inode, its child CL inode, and the target CL leaf typically reside on non-contiguous cache lines.

The hardware prefetcher cannot predict these indirect jumps (each CL internal node stores child slot indices that must be read before the next cache line address is known). This creates a dependent chain of cache misses: *read child index* → *compute address* → *miss on that address* → *repeat.*

At N=4M with random inserts, the 4 KiB pages are spread across a ∼20 MiB working set, exceeding L2 but fitting in L3. Each page access brings the header cache line into L1, but the CL sub-nodes deeper in the page have been evicted since the last visit to that page.

9.2 Optimisation: Software Prefetching

Software prefetch hints (`__builtin_prefetch`) have been added at three levels:

`page_find_leaf` (CL sub-tree descent)
After reading the page header's `root_slot`, the root CL node's cache line is prefetched before the loop begins. Within each loop iteration, after determining the child slot index from `cl_inode_search`, the child's cache line is prefetched before the next iteration accesses it. This converts the dependent miss chain into overlapped prefetch + computation.

`find_leaf` (outer B+ tree descent)
After `mt_inode_search` determines the child pointer, the first cache line of the child node is prefetched. For internal-to-internal transitions, this warms the next inode's header and first keys. For the last level (internal-to-leaf), this warms the page header so that `page_find_leaf` starts with the header in L2.

`mt_inode_search` (binary search in outer inodes)
For nodes exceeding the SIMD linear scan threshold, the binary search prefetches the midpoints of both candidate halves before each comparison. This converts the $O(\log N)$ dependent miss chain of binary search into a pipelined prefetch sequence.

All prefetches use `locality=1` (L2 hint), appropriate for data that will be used once in the near future but is unlikely to be reused before eviction.

9.3 Optimisation: Pointer Tagging for CL Root Prefetch

Instruction-level profiling revealed that the initial software prefetching approach had insufficient lead time: the CL root prefetch inside `page_find_leaf` was issued *after* loading the page header (to read `root_slot`), leaving only ∼5 ns of computation before the CL root access—far less than the ∼100 ns needed for an L3 miss to resolve.

To address this, leaf metadata is embedded in the low bits of child pointers in the outer B+ tree's internal nodes. All node pointers are 4096-byte aligned, providing 12 guaranteed-zero low bits. The encoding is:

Bits	Field	Range
0–5	`root_slot`	1–63 (CL sub-tree root slot index)
6–8	`sub_height`	0–7 (CL sub-tree height)
9–11	*reserved*	(available for future use, e.g. fullness counts)

With this scheme, `find_leaf` can extract `root_slot` from the tagged child pointer *without* reading the page header, enabling it to issue *two* prefetches simultaneously at the last outer-tree level:

1. The page header cache line (slot 0), as before;
2. The CL root cache line (`slots[root_slot − 1]`), computed from the tag.

Both prefetches issue at the same time, overlapping with whatever inode search computation follows. Tags are maintained at leaf creation (bulk-load, split) and updated after each insert or delete that may change `root_slot` or `sub_height`. Every child pointer read goes through an `mt_untag()` mask operation (a single AND instruction) to strip the low 12 bits before dereferencing. Stale tags on cold paths (rebalance) merely cause a wasted prefetch—never a correctness issue.

10 Detailed Results Table

Matryoshka rows highlighted in blue.

Table 5: Full benchmark results.

Library	Workload	N	Mop/s	ns/op
abseil_btree	mixed	65,536	5.47	182.8
abseil_btree	mixed	262,144	5.42	184.5
abseil_btree	mixed	1,048,576	4.94	202.6
abseil_btree	mixed	4,194,304	3.23	309.9
abseil_btree	mixed	16,777,216	1.88	533.1
abseil_btree	rand_delete	65,536	3.56	280.8
abseil_btree	rand_delete	262,144	3.77	265.4
abseil_btree	rand_delete	1,048,576	3.25	307.6
abseil_btree	rand_delete	4,194,304	1.95	514.1
abseil_btree	rand_delete	16,777,216	1.61	619.9
abseil_btree	rand_insert	65,536	4.15	241.2
abseil_btree	rand_insert	262,144	3.80	263.0
abseil_btree	rand_insert	1,048,576	2.89	345.8
abseil_btree	rand_insert	4,194,304	2.11	473.7
abseil_btree	rand_insert	16,777,216	1.43	698.7
abseil_btree	search_after_churn	65,536	3.60	277.9
abseil_btree	search_after_churn	262,144	2.73	365.6
abseil_btree	search_after_churn	1,048,576	2.09	479.4
abseil_btree	search_after_churn	4,194,304	1.38	725.3
abseil_btree	search_after_churn	16,777,216	1.23	813.8
abseil_btree	seq_insert	65,536	7.67	130.4
abseil_btree	seq_insert	262,144	7.86	127.2
abseil_btree	seq_insert	1,048,576	7.77	128.7
abseil_btree	seq_insert	4,194,304	8.51	117.4
abseil_btree	seq_insert	16,777,216	8.44	118.4
abseil_btree	ycsb_a	65,536	7.66	130.6
abseil_btree	ycsb_a	262,144	6.61	151.4
abseil_btree	ycsb_a	1,048,576	7.51	133.1
abseil_btree	ycsb_a	4,194,304	7.30	136.9
abseil_btree	ycsb_a	16,777,216	6.25	160.1
abseil_btree	ycsb_b	65,536	3.87	258.2
abseil_btree	ycsb_b	262,144	3.16	316.5
abseil_btree	ycsb_b	1,048,576	3.25	307.6
abseil_btree	ycsb_b	4,194,304	1.39	719.3
abseil_btree	ycsb_b	16,777,216	0.97	1,036.2
libart	mixed	65,536	9.85	101.6
libart	mixed	262,144	4.90	204.2
libart	mixed	1,048,576	4.01	249.6
libart	mixed	4,194,304	3.27	306.1
libart	mixed	16,777,216	2.61	382.9
libart	rand_delete	65,536	6.53	153.1
libart	rand_delete	262,144	7.19	139.0
libart	rand_delete	1,048,576	3.71	269.9
libart	rand_delete	4,194,304	2.29	437.3
libart	rand_delete	16,777,216	1.94	514.8
libart	rand_insert	65,536	6.47	154.6

Continued on next page

Table 5: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
libart	rand_insert	262,144	6.20	161.2
libart	rand_insert	1,048,576	4.79	209.0
libart	rand_insert	4,194,304	3.57	280.2
libart	rand_insert	16,777,216	2.58	387.5
libart	search_after_churn	65,536	13.16	76.0
libart	search_after_churn	262,144	10.88	91.9
libart	search_after_churn	1,048,576	6.94	144.0
libart	search_after_churn	4,194,304	6.23	160.4
libart	search_after_churn	16,777,216	4.16	240.4
libart	seq_insert	65,536	8.17	122.4
libart	seq_insert	262,144	8.75	114.3
libart	seq_insert	1,048,576	8.67	115.3
libart	seq_insert	4,194,304	9.76	102.5
libart	seq_insert	16,777,216	9.14	109.3
libart	ycsb_a	65,536	8.61	116.1
libart	ycsb_a	262,144	9.07	110.2
libart	ycsb_a	1,048,576	8.21	121.8
libart	ycsb_a	4,194,304	8.29	120.7
libart	ycsb_a	16,777,216	7.59	131.8
libart	ycsb_b	65,536	12.51	79.9
libart	ycsb_b	262,144	10.22	97.8
libart	ycsb_b	1,048,576	5.95	168.1
libart	ycsb_b	4,194,304	4.21	237.6
libart	ycsb_b	16,777,216	3.03	330.4
matryoshka	mixed	65,536	4.67	214.2
matryoshka	mixed	262,144	3.79	263.6
matryoshka	mixed	1,048,576	2.72	367.4
matryoshka	mixed	4,194,304	2.00	500.7
matryoshka	mixed	16,777,216	2.10	475.6
matryoshka	rand_delete	65,536	3.22	310.8
matryoshka	rand_delete	262,144	2.89	346.6
matryoshka	rand_delete	1,048,576	2.08	481.0
matryoshka	rand_delete	4,194,304	1.85	539.7
matryoshka	rand_delete	16,777,216	1.20	834.9
matryoshka	rand_insert	65,536	3.91	255.9
matryoshka	rand_insert	262,144	3.55	282.0
matryoshka	rand_insert	1,048,576	2.56	391.4
matryoshka	rand_insert	4,194,304	1.84	543.0
matryoshka	rand_insert	16,777,216	1.38	724.5
matryoshka	search_after_churn	65,536	5.09	196.3
matryoshka	search_after_churn	262,144	3.89	256.9
matryoshka	search_after_churn	1,048,576	3.18	314.7
matryoshka	search_after_churn	4,194,304	2.45	408.4
matryoshka	search_after_churn	16,777,216	1.43	697.7
matryoshka	seq_insert	65,536	9.76	102.5
matryoshka	seq_insert	262,144	9.66	103.6

Continued on next page

Table 5: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
matryoshka	seq_insert	1,048,576	8.90	112.4
matryoshka	seq_insert	4,194,304	8.26	121.1
matryoshka	seq_insert	16,777,216	7.12	140.5
matryoshka	ycsb_a	65,536	9.18	108.9
matryoshka	ycsb_a	262,144	9.26	108.0
matryoshka	ycsb_a	1,048,576	8.30	120.4
matryoshka	ycsb_a	4,194,304	6.28	159.2
matryoshka	ycsb_a	16,777,216	5.76	173.6
matryoshka	ycsb_b	65,536	4.72	212.0
matryoshka	ycsb_b	262,144	4.08	245.1
matryoshka	ycsb_b	1,048,576	3.20	312.0
matryoshka	ycsb_b	4,194,304	2.03	493.6
matryoshka	ycsb_b	16,777,216	1.78	561.6
std_set	mixed	65,536	2.57	389.5
std_set	mixed	262,144	1.52	657.1
std_set	mixed	1,048,576	0.84	1,194.6
std_set	mixed	4,194,304	0.68	1,471.4
std_set	mixed	16,777,216	0.52	1,934.8
std_set	rand_delete	65,536	2.31	433.2
std_set	rand_delete	262,144	1.37	730.0
std_set	rand_delete	1,048,576	0.62	1,603.7
std_set	rand_delete	4,194,304	0.40	2,485.7
std_set	rand_delete	16,777,216	0.30	3,321.7
std_set	rand_insert	65,536	2.99	334.3
std_set	rand_insert	262,144	2.04	490.3
std_set	rand_insert	1,048,576	0.90	1,107.2
std_set	rand_insert	4,194,304	0.48	2,087.3
std_set	rand_insert	16,777,216	0.37	2,679.0
std_set	search_after_churn	65,536	2.30	433.9
std_set	search_after_churn	262,144	1.10	907.2
std_set	search_after_churn	1,048,576	0.64	1,555.0
std_set	search_after_churn	4,194,304	0.43	2,326.6
std_set	search_after_churn	16,777,216	0.30	3,349.1
std_set	seq_insert	65,536	4.12	242.7
std_set	seq_insert	262,144	2.62	381.4
std_set	seq_insert	1,048,576	2.36	424.5
std_set	seq_insert	4,194,304	1.92	520.8
std_set	seq_insert	16,777,216	1.67	599.4
std_set	ycsb_a	65,536	3.82	261.5
std_set	ycsb_a	262,144	2.22	451.1
std_set	ycsb_a	1,048,576	1.97	507.1
std_set	ycsb_a	4,194,304	1.48	674.7
std_set	ycsb_a	16,777,216	1.23	813.5
std_set	ycsb_b	65,536	2.08	480.9
std_set	ycsb_b	262,144	1.00	997.6
std_set	ycsb_b	1,048,576	0.63	1,592.7

Continued on next page

Table 5: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
std_set	ycsb_b	4,194,304	0.38	2,610.7
std_set	ycsb_b	16,777,216	0.32	3,148.9
tlx_btree	mixed	65,536	4.43	225.6
tlx_btree	mixed	262,144	4.03	248.4
tlx_btree	mixed	1,048,576	2.71	368.7
tlx_btree	mixed	4,194,304	1.95	512.9
tlx_btree	mixed	16,777,216	1.52	658.3
tlx_btree	rand_delete	65,536	2.80	356.8
tlx_btree	rand_delete	262,144	3.01	332.8
tlx_btree	rand_delete	1,048,576	1.91	522.3
tlx_btree	rand_delete	4,194,304	1.24	804.0
tlx_btree	rand_delete	16,777,216	0.99	1,014.3
tlx_btree	rand_insert	65,536	3.17	315.2
tlx_btree	rand_insert	262,144	2.94	339.6
tlx_btree	rand_insert	1,048,576	2.49	401.6
tlx_btree	rand_insert	4,194,304	1.69	592.1
tlx_btree	rand_insert	16,777,216	1.07	937.7
tlx_btree	search_after_churn	65,536	4.04	247.4
tlx_btree	search_after_churn	262,144	3.06	327.3
tlx_btree	search_after_churn	1,048,576	2.28	439.1
tlx_btree	search_after_churn	4,194,304	1.48	675.2
tlx_btree	search_after_churn	16,777,216	1.04	959.4
tlx_btree	seq_insert	65,536	11.26	88.8
tlx_btree	seq_insert	262,144	8.71	114.8
tlx_btree	seq_insert	1,048,576	9.58	104.4
tlx_btree	seq_insert	4,194,304	9.35	107.0
tlx_btree	seq_insert	16,777,216	7.17	139.4
tlx_btree	ycsb_a	65,536	8.78	113.9
tlx_btree	ycsb_a	262,144	8.83	113.2
tlx_btree	ycsb_a	1,048,576	7.58	132.0
tlx_btree	ycsb_a	4,194,304	6.60	151.6
tlx_btree	ycsb_a	16,777,216	5.38	186.0
tlx_btree	ycsb_b	65,536	1.99	502.5
tlx_btree	ycsb_b	262,144	2.17	460.4
tlx_btree	ycsb_b	1,048,576	2.10	475.3
tlx_btree	ycsb_b	4,194,304	1.29	776.9
tlx_btree	ycsb_b	16,777,216	0.91	1,104.6

11 Analysis and Diagnosis

11.1 Benchmark Access Patterns

The seven workloads exercise distinct access patterns that interact differently with each data structure's memory layout:

`seq_insert`

Keys arrive in ascending order (1, 3, 5, . . .). Structures with append-optimised paths benefit: B-trees fill leaves left-to-right with no splits until full; red-black trees rebalance but maintain temporal locality in the allocator. ART builds monotonically deeper paths in byte-order.

Matryoshka benefits from sequential CL sub-node filling within each page, but the nested sub-tree overhead (navigating CL internals) makes each insert more expensive than a simple sorted-array append.

`rand_insert`
A Fisher–Yates shuffle of $[0, N)$ scaled to odd values. Every insert touches a random leaf, maximising cache pressure. This is the workload most sensitive to node size and memory layout: wide B-tree leaves amortise random access; pointer-chasing structures (`std::set`) incur a cache miss per tree level; ART's fixed-depth (4-byte key $\Rightarrow \leq 4$ levels) limits misses.

`rand_delete`
Bulk-loads N sorted keys, then deletes all in shuffled order. The bulk-load gives every structure its optimal starting layout. Random deletion stresses rebalancing: red-black trees do O(1) rotations; B-trees `memmove` within leaves and occasionally merge or redistribute; matryoshka merges CL sub-nodes within a page, with page-level merges only on underflow.

`mixed`
Alternating insert (new key beyond current max) and delete (random existing key) on a pre-loaded tree. This creates a steady-state workload where the tree size fluctuates around N. Structures that handle interleaved modifications without excessive rebalancing perform well.

`ycsb_a` (95% insert / 5% search)
Simulates a write-dominated OLTP workload. Inserts are sequential (monotonically increasing keys), so append-path efficiency matters. The 5% predecessor searches touch recently-inserted regions, favouring structures with good temporal locality in their leaf layer.

`ycsb_b` (50% delete / 50% search)
Deletes from a pre-loaded tree interleaved with random predecessor searches. The shrinking tree tests whether structures maintain search efficiency as occupancy drops and structural changes (merges, rebalancing) occur.

`search_after_churn`
Pure search workload on a tree that has undergone insert/delete churn (untimed). 5,000,000 random predecessor searches isolate search throughput from modification cost. This workload is most sensitive to in-leaf search cost and memory layout, not structural operations. Note: ART lacks native predecessor search; its wrapper falls back to point lookup, giving it an advantage on hit rate but testing a different operation.

All workloads use 4-byte `int32_t` keys encoded as odd values ($2i+1$), generated by xorshift64 with fixed seeds for reproducibility.

11.2 Matryoshka: Nested Sub-Tree Tradeoffs

Each insert or delete navigates the page-level CL sub-tree (2–3 SIMD comparisons on 12–15 keys per level) to a target CL leaf, then performs a scalar `memmove` of at most 14 keys within that 64 B cache-line sub-node. The cost per modification is $O(h_s \times b)$ where $h_s \leq 2$ is the sub-tree height and $b = 15$ is the CL leaf capacity—roughly 30–45 key touches, all within a single 4 KiB page.

CL sub-node splits and merges occur only when a CL leaf overflows (15 keys) or underflows (< 7 keys). Page-level splits occur only when all 63 CL slots are exhausted ($\sim$855 keys/page).

The nested design adds a constant overhead per operation compared to flat sorted-array B-tree leaves, where a single `memmove` suffices. At N=1,048,576: matryoshka achieves 2.56 Mop/s on random insert, vs. 2.89 (Abseil) and 2.49 (TLX).

However, SIMD search through the CL sub-tree is used during both search *and* the navigation phase of insert/delete. This yields search-after-churn throughput of 3.18 Mop/s at N=1,048,576. The key advantage is that modifications touch a single cache-line sub-node rather than shifting an entire sorted leaf array.

11.3 Detailed Comparison: std::set (Red-Black Tree)

`std::set` uses a balanced binary search tree with one heap-allocated node per key (40–48 B on 64-bit systems: two child pointers, parent pointer, colour bit, key, allocator overhead).

Insert and delete. At N=16,777,216: 0.37 Mop/s (random insert) and 0.30 Mop/s (random delete)—the slowest of all libraries tested. Each operation traverses $O(\log_2 N)$ levels with a pointer dereference (and likely cache miss) at every level. The 40–48 B node size means ∼1 useful key per cache line, so every level is a full cache miss for large N.

Sequential insert. At 1.67 Mop/s (N=16,777,216), sequential insert is only modestly better than random because the allocator provides some temporal locality, but the red-black tree still requires $O(\log N)$ pointer chases and rotations.

Search. Search-after-churn: 0.30 Mop/s. Binary search through $\log_2 N \approx 24$ levels of pointer chasing is inherently cache-unfriendly. `std::set` has no mechanism for SIMD-accelerated search or cache-line-aware layout.

Scaling. `std::set` shows the steepest throughput degradation from small to large N (random insert scales 8.0× from N=65,536 to N=16,777,216) because the working set of pointer-chased nodes quickly exceeds cache capacity.

11.4 Detailed Comparison: TLX btree_set

TLX implements a B+ tree with sorted-array leaves. Leaf capacity is typically 64–128 keys (depends on template parameters and key size). Internal nodes use sorted arrays of separator keys with binary search.

Insert and delete. At N=16,777,216: 1.07 Mop/s (random insert) and 0.99 Mop/s (random delete). Each leaf insert is a binary search followed by a `memmove` of the leaf's sorted array. The average shift is $B/2 \approx$ 32–64 keys per insert, but the entire operation stays within one or two cache lines for small leaves.

Search. Search-after-churn: 1.04 Mop/s. TLX uses scalar binary search within leaves, which incurs $\lceil \log_2 B \rceil$ comparisons with data-dependent branches. This is slower than SIMD linear scan for the same leaf size.

Sequential insert. 7.17 Mop/s. The B+ tree append path is efficient: new keys land at the rightmost leaf with minimal shifting, and splits propagate only when the leaf is full.

Comparison to matryoshka. TLX's flat sorted-array leaves have a lower constant factor per insert (one `memmove` vs. CL sub-tree navigation), but matryoshka's wider pages (855 keys vs. ∼128) reduce the outer tree height and number of leaf splits. At large N, the outer-tree traversal cost dominates, and matryoshka's SIMD-accelerated outer internal node search closes the gap.

11.5 Detailed Comparison: Abseil btree_set

Abseil's B-tree uses a similar sorted-array design to TLX but with different node sizes and allocation strategies. Leaf nodes hold up to ∼256 keys in a single sorted array.

Insert and delete. At N=16,777,216: 1.43 Mop/s (random insert) and 1.61 Mop/s (random delete). The wider leaves mean fewer tree levels and splits, but each `memmove` within a leaf shifts more keys on average ($B/2 \approx 128$).

Search. Search-after-churn: 1.23 Mop/s. Abseil uses scalar binary search within leaves. The wider leaves reduce tree height (fewer pointer chases) but increase the number of comparisons per leaf ($\lceil \log_2 256 \rceil = 8$ vs. $\lceil \log_2 128 \rceil = 7$ for TLX).

TLX vs. Abseil. On random insert at N=16,777,216, TLX and Abseil are within 25% of each other. Abseil's wider leaves trade cheaper outer traversal (fewer levels) for more expensive in-leaf operations (larger `memmove`). The two designs converge in throughput because the cache miss cost of locating the target leaf dominates at large N.

11.6 Detailed Comparison: libart (Adaptive Radix Tree)

ART uses a radix/trie structure with adaptive node types (Node4, Node16, Node48, Node256) that compact sparse levels. For 4-byte keys, the tree has at most 4 levels regardless of N.

Insert and delete. At N=16,777,216: 2.58 Mop/s (random insert) and 1.94 Mop/s (random delete). ART's $O(k)$ complexity (key length, not tree size) means insert cost is nearly constant across dataset sizes. The scaling ratio from N=65,536 to N=16,777,216 is 2.5× —the flattest of all structures tested.

Search. Search-after-churn: 4.16 Mop/s. ART achieves the highest absolute search throughput because its point lookups traverse ≤ 4 levels, each requiring a single indexed array access (no comparison-based search within nodes for Node256). *However*, the benchmark uses point lookups for ART rather than predecessor search (which ART does not natively support), so this comparison is not apples-to-apples with the other structures.

Access pattern interaction. ART's per-byte radix decomposition means key distribution matters less than key length. The uniform random keys in these benchmarks create well-distributed tries with few path-compressed nodes. A workload with clustered keys sharing long common prefixes would trigger more path compression and potentially different performance characteristics.

Memory overhead. ART's adaptive node types (4, 16, 48, or 256 children) trade memory for access speed. At high occupancy, most internal nodes are Node48 or Node256, using 256–2048 B per node regardless of actual fan-out—significantly more memory per key than B-tree or matryoshka designs.

11.7 Hardware Counter Comparison

Table 6: Hardware counters across all libraries (rand_insert, N=16,777,216).

Library	**Cache Miss (%)**	**L1d Miss (%)**	**LLC Miss (%)**	**dTLB Miss (/1K ops)**	**Branch Miss (%)**	**IPC**
`matryoshka`	73.2	4.2	22.0	2.8	10.0	0.56
`std_set`	67.7	13.6	44.2	66.9	3.0	0.14
`tlx_btree`	64.9	4.9	31.9	16.7	10.4	0.46
`libart`	64.8	7.5	42.6	35.5	1.2	0.60
`abseil_btree`	63.3	6.9	33.4	16.4	7.1	0.64

The hardware counters reveal distinct micro-architectural profiles:

dTLB pressure. Matryoshka's arena allocator (2 MiB hugepage-backed regions) yields the lowest dTLB miss rate (2.8/1,000 ops). `std::set` has the highest (66.9/1,000) because each pointer chase to a heap-allocated node potentially touches a new 4 KiB page. The B-tree libraries (TLX, Abseil) fall between: their wider nodes reduce the number of distinct pages accessed per operation, but they use standard `malloc` without hugepage awareness. ART's adaptive nodes are heap-allocated but fewer in number than red-black tree nodes, yielding moderate dTLB pressure.

LLC misses. At N=16,777,216 (random insert), all structures exceed LLC capacity. Matryoshka's LLC miss rate is 219.7/1,000 ops vs. 441.8 for `std::set`. Matryoshka confines each leaf operation to a single 4 KiB page (or a few adjacent cache lines for CL sub-node navigation), while red-black tree operations touch $\log_2 N$ widely-scattered nodes. The B-tree libraries achieve comparable LLC rates because their wide leaves also localise work.

IPC. Matryoshka achieves IPC of 0.56 via SIMD search (`_mm_cmpgt_epi32`/`_mm_movemask_ps`) at every level, avoiding data-dependent branches. `std::set`'s pointer-chasing and branch-heavy traversal yields the lowest IPC. ART's indexed array lookups (no comparisons for Node256) can achieve high IPC on cache-hot paths.

Branch misprediction. SIMD mask arithmetic in matryoshka replaces conditional branches, yielding low branch misprediction rates. `std::set` and the scalar-binary-search B-trees (TLX, Abseil) have higher misprediction rates because each comparison is a data-dependent branch. ART's indexed-lookup approach avoids comparison branches within nodes but has conditional branches for node type dispatch.

11.8 Access Pattern Interactions with Data Structure Layout

The interaction between access pattern and memory layout explains much of the performance variation:

Sequential vs. random insert. Sequential insert favours structures with efficient append paths. All B-tree variants (matryoshka, TLX, Abseil) benefit because new keys land at the rightmost leaf. `std::set` benefits less because red-black rebalancing is oblivious to key order. The throughput ratio (seq/rand) at N=16,777,216 reveals how much each structure benefits from locality: matryoshka 4.27× vs. std::set on sequential, 3.70× on random.

Delete after bulk-load vs. interleaved. The `rand_delete` workload starts from a bulk-loaded (optimally packed) tree, giving every structure its best starting point. The `mixed` workload, by contrast, operates on a tree that is continuously modified, creating internal fragmentation. Structures that maintain good occupancy under churn (B-trees with merge/redistribute) degrade less between these workloads than structures with per-node allocation (`std::set`).

The 4 KiB page boundary. Matryoshka's 4 KiB page leaves are sized to match the OS page size, ensuring that navigating the CL sub-tree within a leaf never crosses a page boundary. TLX and Abseil leaves are smaller (<1 KiB), so multiple leaves may share a page—good for spatial locality of adjacent leaves, but each leaf may straddle two cache lines for the `memmove` operation. `std::set` nodes are scattered across the heap with no page-alignment guarantees.

SIMD and branch prediction. Matryoshka's SIMD search produces a bit mask rather than a conditional branch, making it prediction-friendly. The sorted-array B-trees (TLX, Abseil) use scalar binary search with $O(\log B)$ data-dependent branches per leaf, which the branch predictor struggles with for uniform random keys (50/50 taken probability at each comparison).

11.9 Proposed Additional Access Patterns

Several workloads not currently benchmarked would reveal different performance relationships:

Zipfian (skewed) insert
A Zipfian distribution concentrates inserts on a small number of "hot" leaves. B-tree variants would benefit from cache-hot leaves; `std::set` would benefit from a cache-hot path of recently accessed nodes. Matryoshka's per-page CL sub-tree might show more frequent CL splits under concentrated load.

Range scan after insert
Iterate over a range of k keys (e.g. $k = 1000$) after building the tree. Matryoshka's linked leaf pages and dense packing should excel; `std::set`'s in-order traversal via parent pointers would lag. This would highlight the spatial locality advantage of contiguous leaf storage.

Interleaved point lookup and insert
A read-modify-write pattern ("contains then insert if absent") would test whether search and insert share cache-hot state. Matryoshka's search and insert paths share the same CL sub-tree navigation code, so a just-searched path remains cache-hot for the subsequent insert.

Large-key workload
Keys longer than 4 bytes (e.g. 16- or 32-byte strings) would stress ART's strength (key-length-dependent, not N-dependent traversal) while increasing matryoshka's CL sub-node overhead (fewer keys per 64 B cache line). B-tree `memmove` cost would grow linearly with key size.

Delete-heavy with searches (YCSB-D)
A workload where the tree shrinks from N to near-empty while servicing read queries. This would stress merge and redistribute paths, and test whether search throughput degrades as the tree becomes sparsely populated. Matryoshka's CL sub-node merge and page-level redistribute are designed for graceful degradation, but extreme sparsity (few keys spread across many pages) could hurt cache utilisation.

Bulk-load comparison
Timing the bulk-load operation itself (currently untimed) would highlight structural differences: matryoshka distributes keys across CL sub-nodes within pages in a single bottom-up pass; TLX and Abseil use repeated insertion; `std::set` has no bulk-load optimisation.

11.10 Overall Assessment

The matryoshka nesting design achieves competitive insert and delete throughput while preserving SIMD-accelerated search at every level of the hierarchy. Each modification touches a single cache-line sub-node rather than rebuilding an entire sorted leaf array. At the largest dataset size (N=16,777,216), matryoshka's throughput is within 1.9× of the best B-tree competitor on insert-heavy workloads and 1.6× on delete-heavy workloads.

The primary cost of the nesting design is a constant-factor overhead per modification (CL sub-tree navigation), which the flat sorted-array B-trees avoid. This overhead is most visible at small N where the outer tree is shallow and the per-operation cost is dominated by in-leaf work. At large N, where the outer tree traversal and cache misses dominate, the nesting overhead is amortised and matryoshka's SIMD search and dense page layout become decisive advantages.

12 Improvements Since Initial Report

12.1 Superpage-Level Nesting (Implemented)

The nesting now extends to three levels: CL sub-nodes (64 B) within 4 KiB pages within 2 MiB superpages. Each superpage contains a B+ tree of page-level sub-nodes, with page-level internal nodes (681 separator keys, 682 children per 4 KiB page) routing searches to up to 510 page leaves. Maximum capacity per superpage: $510 \times 855 \approx 436$K keys. This confines most operations to a single TLB entry and reduces outer-tree height. Enable via `mt_hierarchy_init_superpage`.

12.2 Wider SIMD: AVX2 and AVX-512 (Implemented)

Compile-time SIMD width selection via `-DMT_SIMD=avx2` or `-DMT_SIMD=avx512`. AVX2 (256-bit) processes 8 keys per comparison in CL leaf predecessor search, CL internal search, and outer internal node search. AVX-512 (512-bit) processes 16 keys per comparison using masked operations. Unaligned loads handle the 4-byte header offset within CL sub-nodes. SSE2 (128-bit, 4 keys) remains the baseline fallback.

12.3 Batch Insert and Delete API (Implemented)

`matryoshka_insert_batch(tree, keys, n)` and `matryoshka_delete_batch(tree, keys, n)` sort incoming keys, group them by target leaf, and amortise outer-tree traversal across each group. On page-full or underflow, the path is re-navigated for remaining keys. Both functions work with page leaves and superpages.

12.4 Future: Variable-Length Keys

The current 4-byte `int32_t` key format could be extended to variable-length keys by storing key offsets or using indirection within CL sub-nodes. This would broaden applicability at the cost of some cache efficiency.

References

[1] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. *FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs.* SIGMOD '10, pp. 339–350, 2010.

[2] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indexes.* Acta Informatica, 1(3):173–189, 1972.

[3] V. Leis, A. Kemper, and T. Neumann. *The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases.* ICDE '13, pp. 38–49, 2013.

[4] J. Rao and K. A. Ross. *Making B+-Trees Cache Conscious in Main Memory.* SIGMOD '00, pp. 475–486, 2000.