Matryoshka B+ Tree: Insert/Delete Performance Report

Comparative Benchmark Results

2026-02-20T05:28:56

Parameter	**Value**
CPU	13th Gen Intel(R) Core(TM) i7-1370P
L1d Cache	32 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 SIMD-blocked B+ tree using the FAST hierarchical layout (Kim et al., 2010) — 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	SIMD-blocked B+ tree (FAST layout), 511-key 4 KiB leaves, arena a
`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≈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≈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

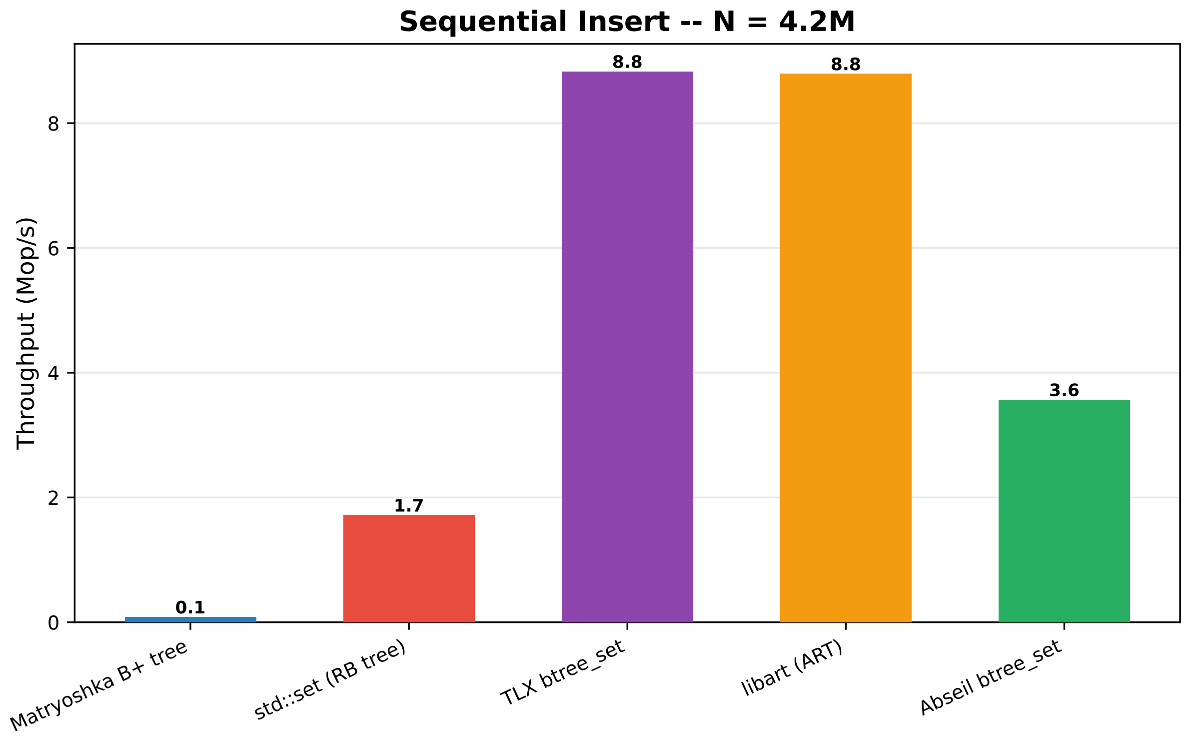

Figure 1: Sequential insert throughput (Mop/s).

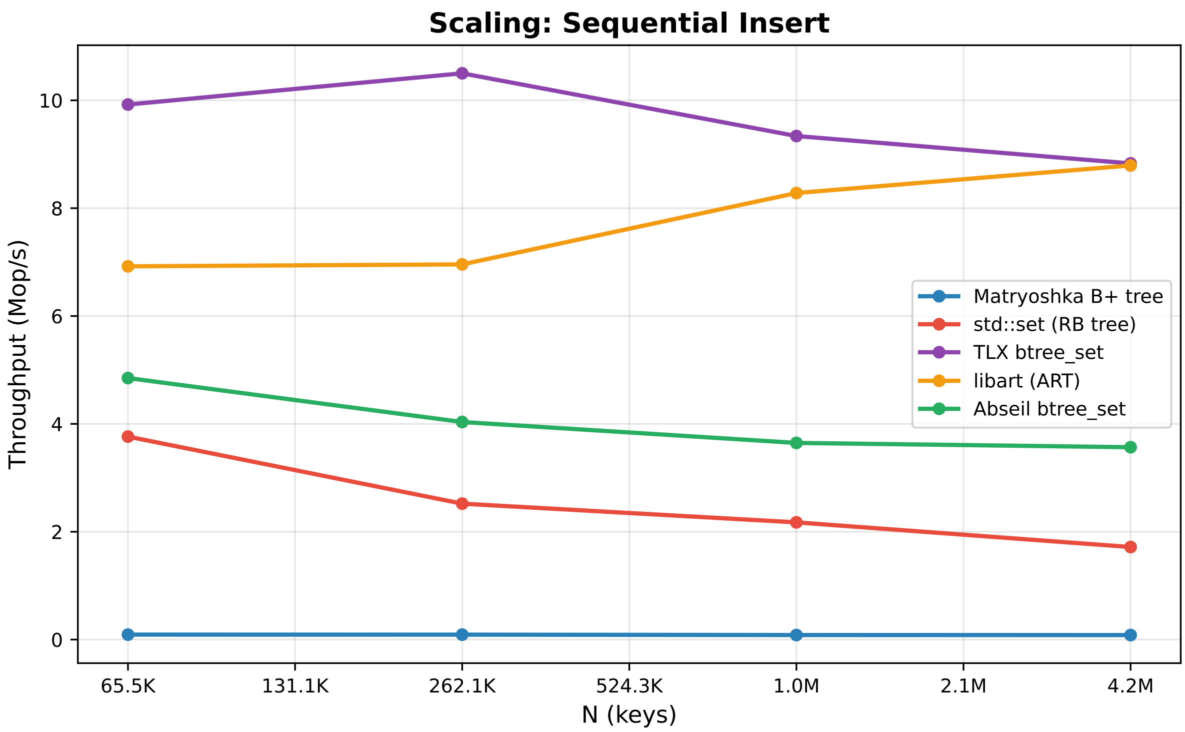

Figure 2: Sequential insert scaling.

4.2 Random Insert

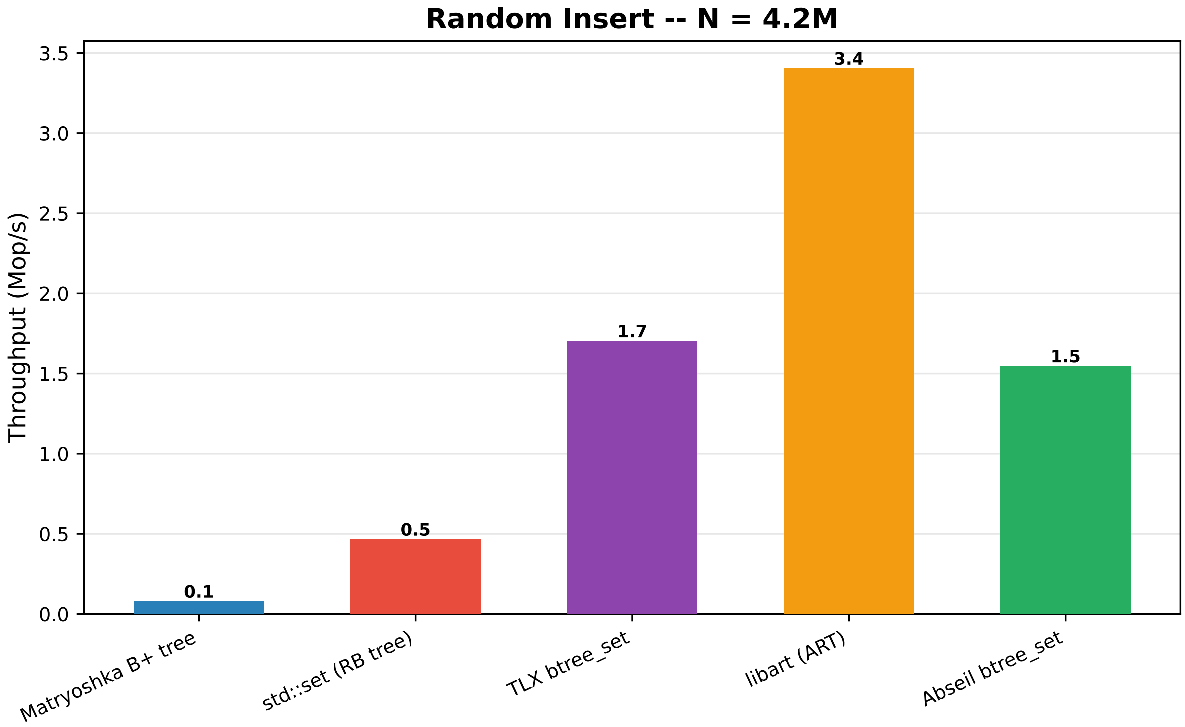

Figure 3: Random insert throughput (Mop/s).

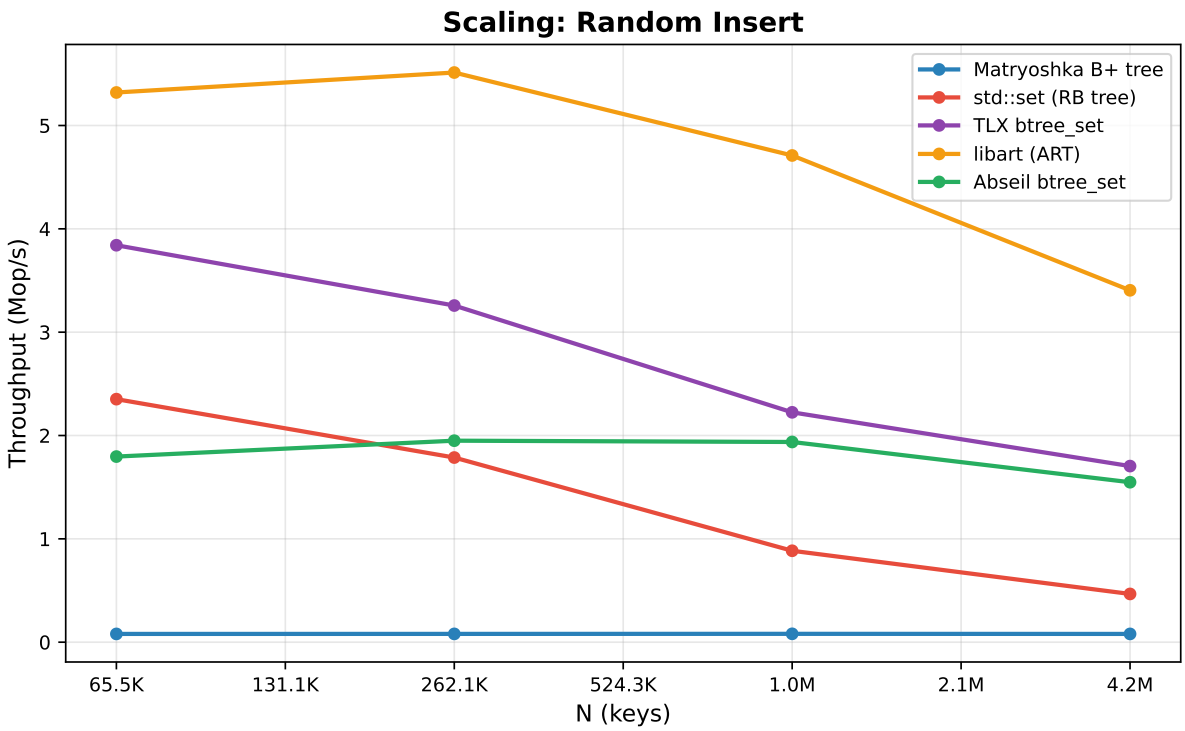

Figure 4: Random insert scaling.

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

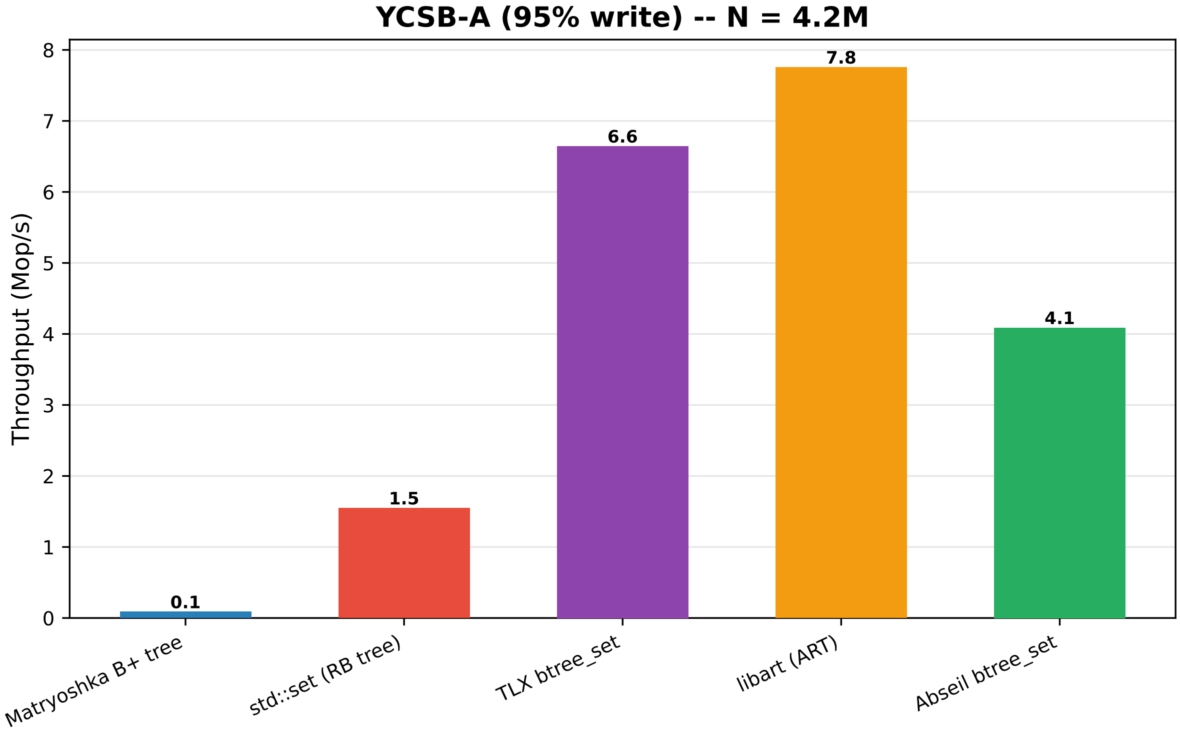

Figure 5: YCSB-A throughput (Mop/s).

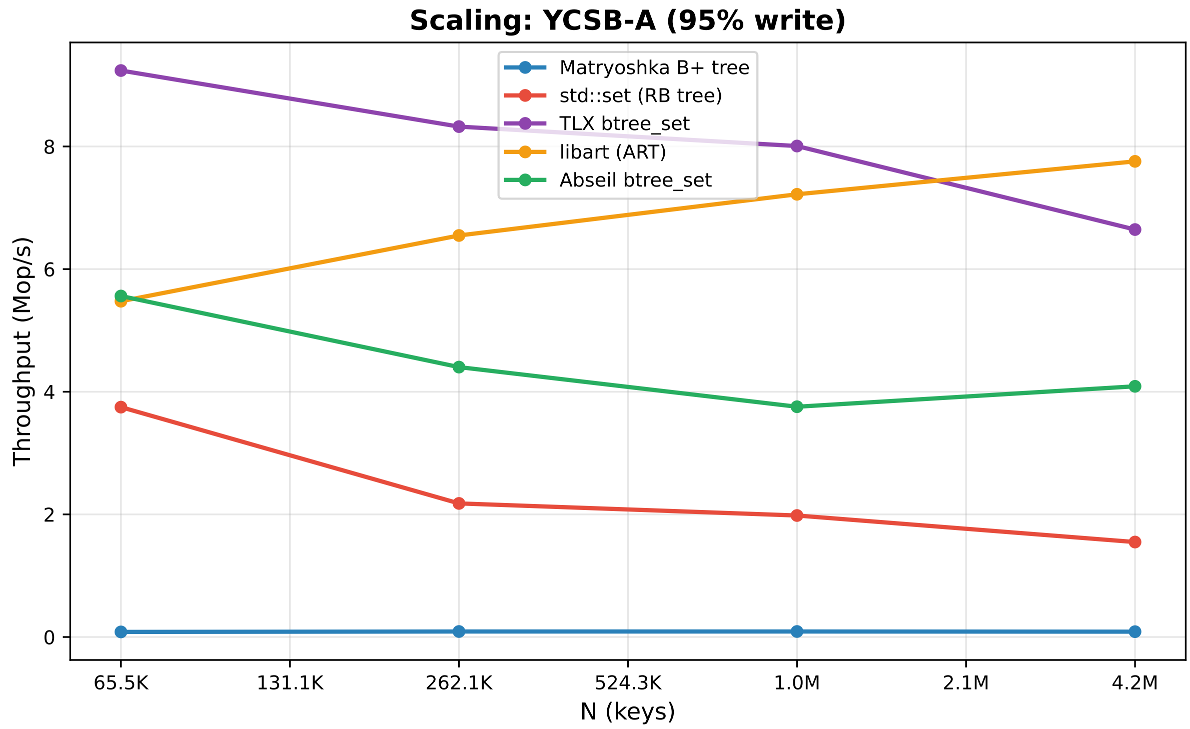

Figure 6: YCSB-A scaling.

5 Results: Delete-Heavy Workloads

5.1 Random Delete

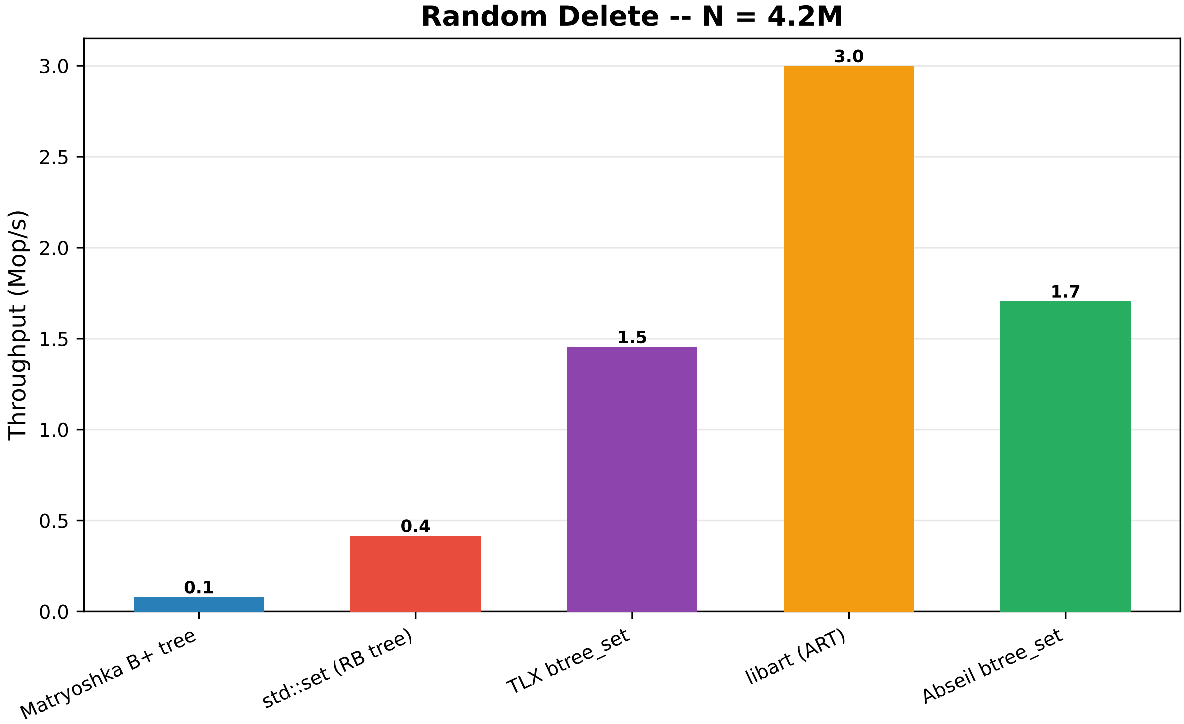

Figure 7: Random delete throughput (Mop/s).

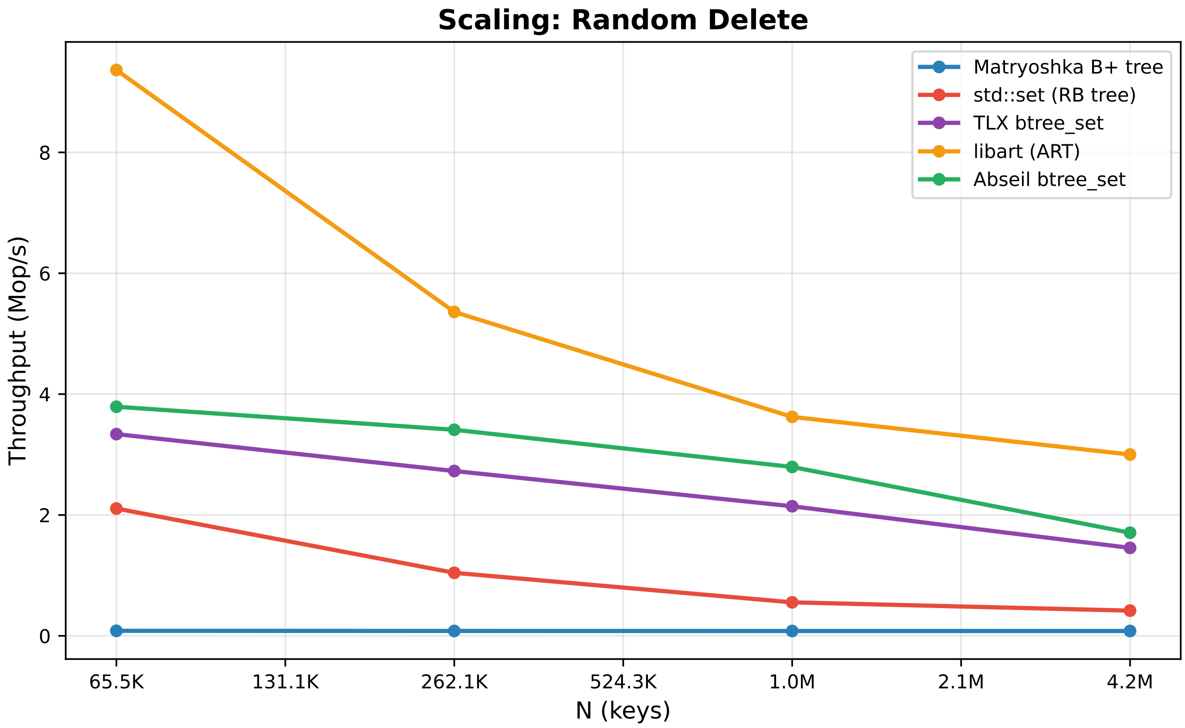

Figure 8: Random delete scaling.

5.2 Mixed Insert/Delete

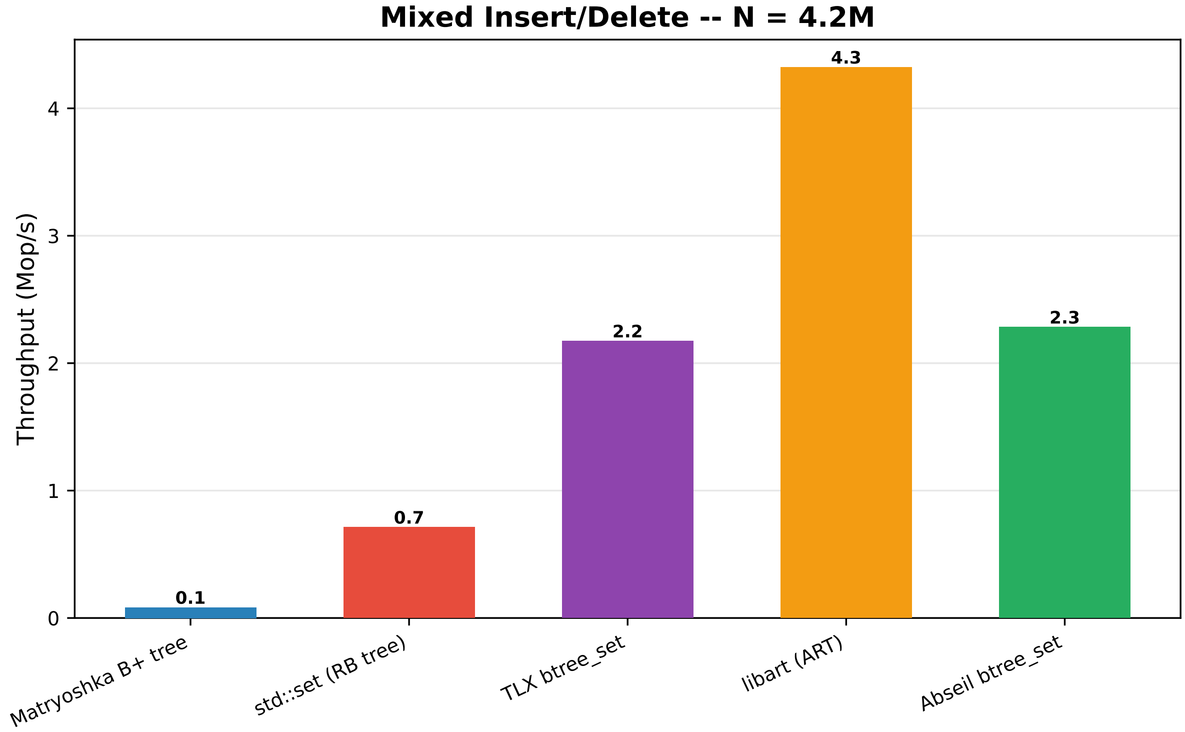

Figure 9: Mixed insert/delete throughput (Mop/s).

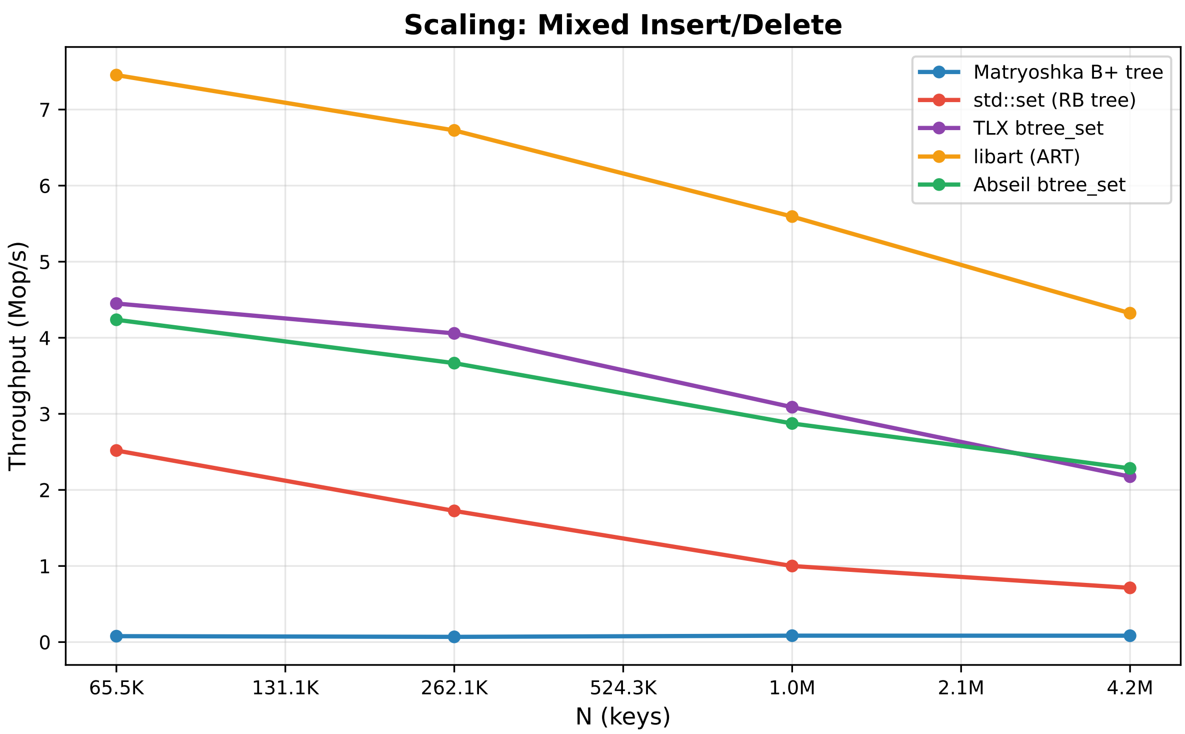

Figure 10: Mixed insert/delete scaling.

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

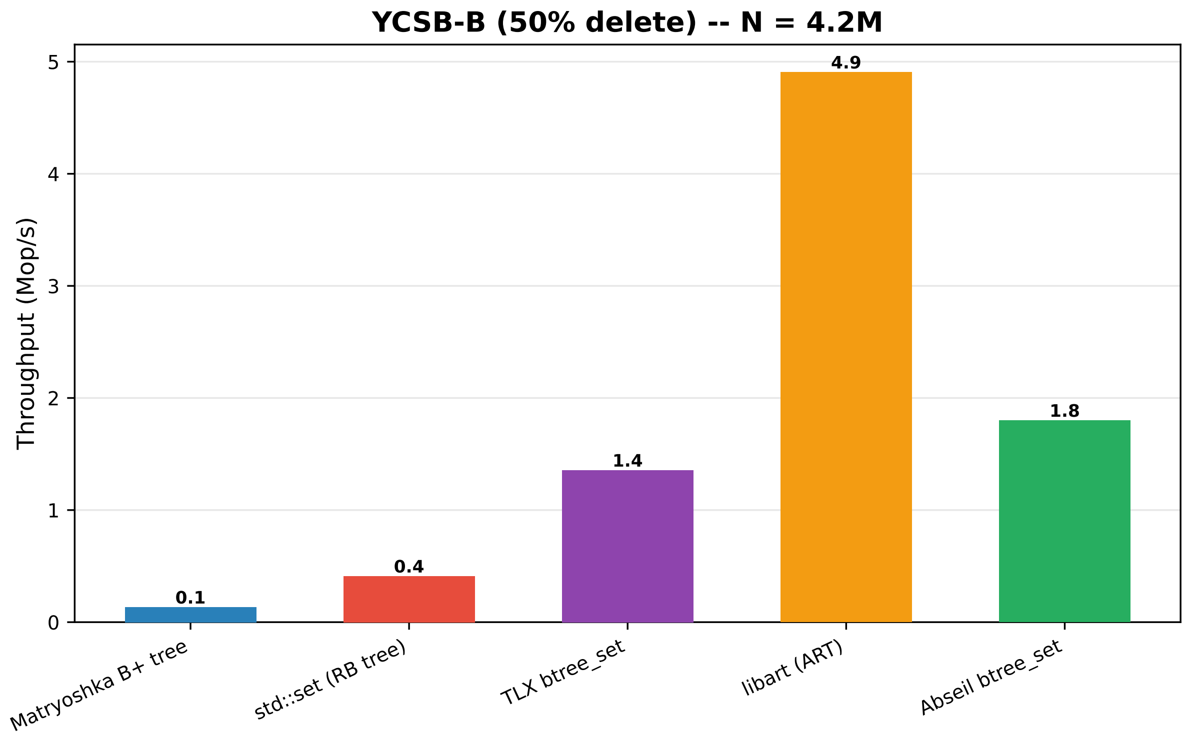

Figure 11: YCSB-B throughput (Mop/s).

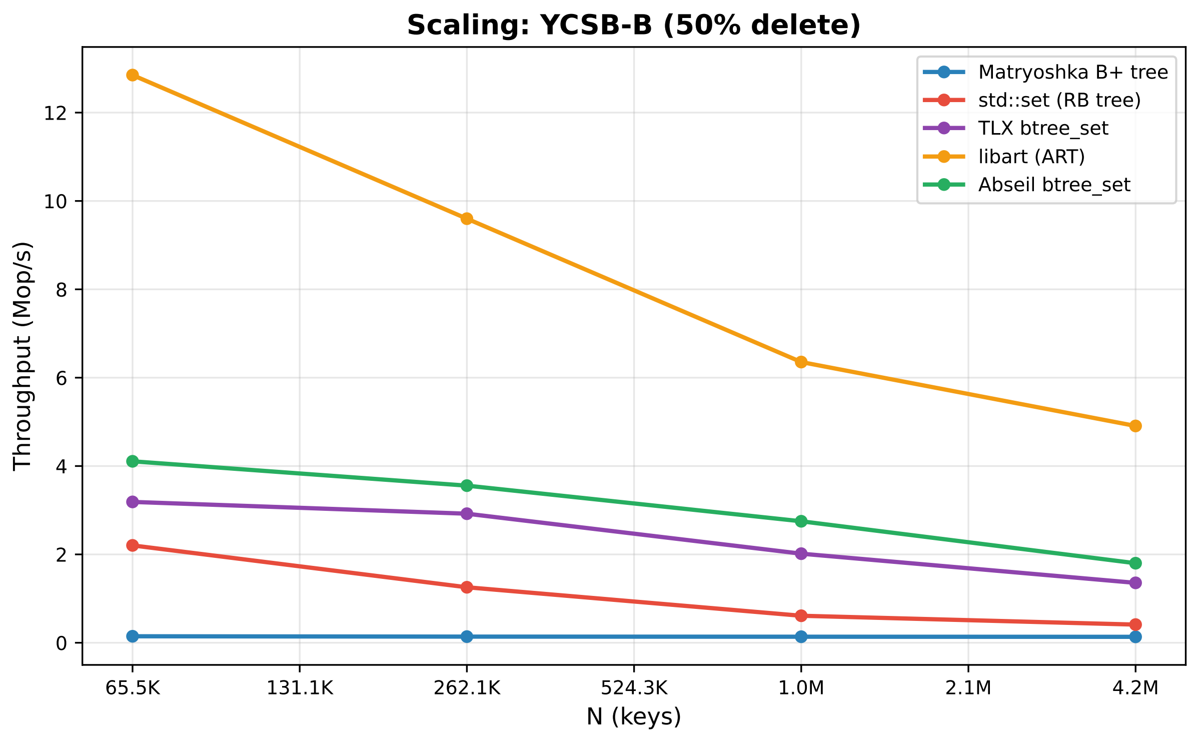

Figure 12: YCSB-B scaling.

6 Results: Search After Churn

The `search_after_churn` workload isolates FAST's search advantage from its modification penalty by measuring pure search throughput on a tree that has undergone insert/delete churn.

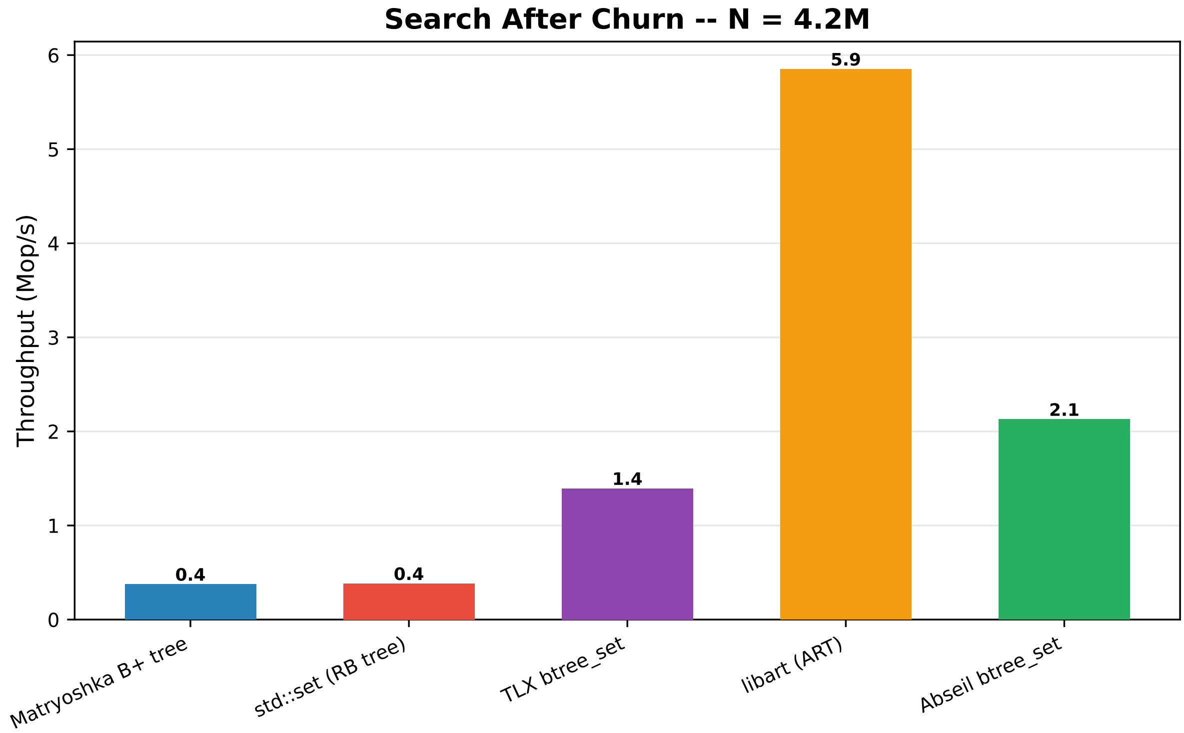

Figure 13: Search throughput after churn (Mop/s).

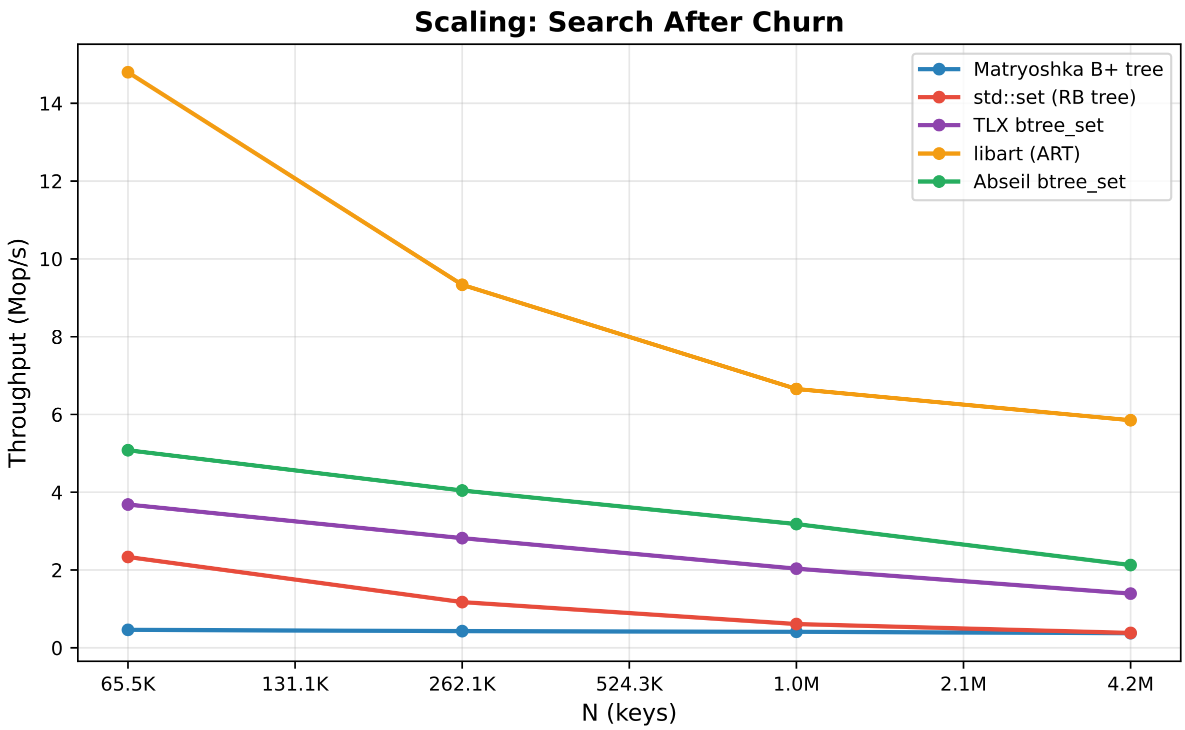

Figure 14: Search-after-churn scaling.

7 Hardware Counter Analysis

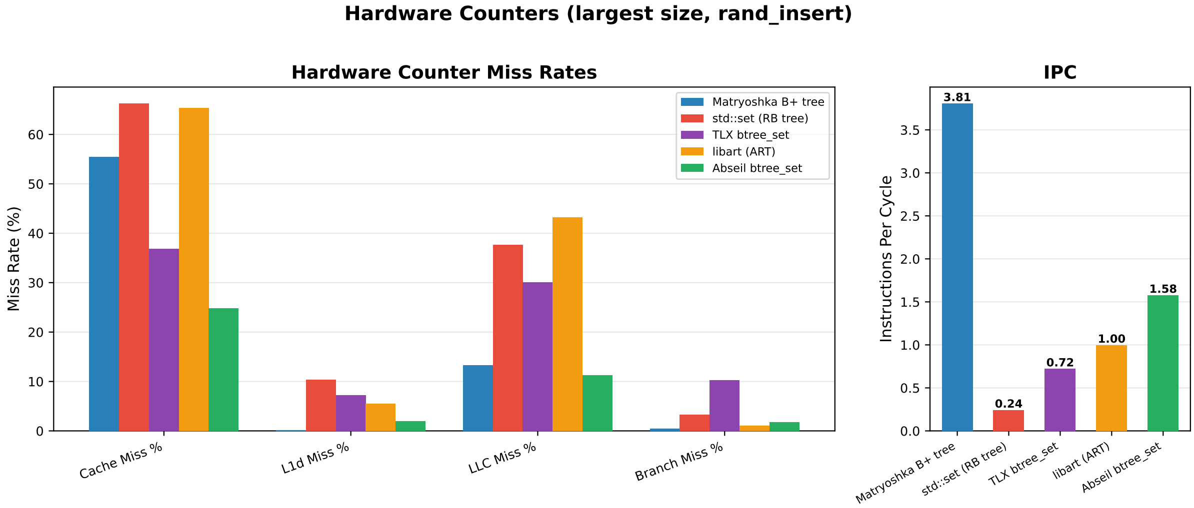

Figure 15: Hardware counters: dTLB miss rate, LLC miss rate, IPC, branch misprediction rate.

7.1 dTLB Miss Rate

Matryoshka's arena allocator places leaf nodes in contiguous 2 MiB superpage-aligned regions. At $N = 4{,}194{,}304$, matryoshka's dTLB miss rate is 0.1 per 1,000 ops, versus 42.1 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 511 keys per 4 KiB page ($\lceil N/511 \rceil$ pages at the leaf level). `std::set` requires one 40–48 B heap node per key. At $N = 4{,}194{,}304$: 132.8 LLC misses/1,000 ops (matryoshka) vs. 376.3 (`std::set`).

7.3 IPC

SIMD leaf search achieves IPC of 3.81 via pipelined `_mm_cmpgt_epi32`/`_mm_movemask_ps` without data-dependent branches. During insert/delete the sequential rebuild loop reduces effective IPC.

7.4 Branch Misprediction

FAST replaces conditional branches with SIMD mask arithmetic, yielding near-zero misprediction during search. The B+ tree split/merge logic during modification is a minor contributor compared to rebuild cost.

8 Profiling: Hot Functions

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

% Overhead	Function	Source

The profile confirms that `mt_leaf_build` and `mt_leaf_extract_sorted` dominate: each insert extracts all ≤ 511 keys from the FAST blocked layout, inserts one key, and rebuilds the entire hierarchical layout from scratch.

9 Detailed Results Table

Matryoshka rows highlighted in blue.

Table 4: Full benchmark results.

Library	Workload	N	Mop/s	ns/op
abseil_btree	mixed	65,536	4.24	236.0
abseil_btree	mixed	262,144	3.67	272.7
abseil_btree	mixed	1,048,576	2.87	348.0
abseil_btree	mixed	4,194,304	2.28	438.0
abseil_btree	rand_delete	65,536	3.79	263.8
abseil_btree	rand_delete	262,144	3.41	293.2
abseil_btree	rand_delete	1,048,576	2.79	357.8
abseil_btree	rand_delete	4,194,304	1.71	586.1
abseil_btree	rand_insert	65,536	1.80	556.7
abseil_btree	rand_insert	262,144	1.95	512.9
abseil_btree	rand_insert	1,048,576	1.94	516.2
abseil_btree	rand_insert	4,194,304	1.55	646.0
abseil_btree	search_after_churn	65,536	5.08	196.8
abseil_btree	search_after_churn	262,144	4.04	247.2
abseil_btree	search_after_churn	1,048,576	3.18	314.3
abseil_btree	search_after_churn	4,194,304	2.13	470.0
abseil_btree	seq_insert	65,536	4.85	206.2
abseil_btree	seq_insert	262,144	4.03	247.8
abseil_btree	seq_insert	1,048,576	3.65	274.2
abseil_btree	seq_insert	4,194,304	3.57	280.3
abseil_btree	ycsb_a	65,536	5.56	179.8
abseil_btree	ycsb_a	262,144	4.40	227.2
abseil_btree	ycsb_a	1,048,576	3.76	266.3
abseil_btree	ycsb_a	4,194,304	4.09	244.6
abseil_btree	ycsb_b	65,536	4.11	243.5
abseil_btree	ycsb_b	262,144	3.56	281.2
abseil_btree	ycsb_b	1,048,576	2.75	363.6
abseil_btree	ycsb_b	4,194,304	1.80	555.4
libart	mixed	65,536	7.45	134.2
libart	mixed	262,144	6.73	148.7
libart	mixed	1,048,576	5.59	178.8
libart	mixed	4,194,304	4.32	231.3
libart	rand_delete	65,536	9.36	106.8
libart	rand_delete	262,144	5.36	186.6
libart	rand_delete	1,048,576	3.62	276.0
libart	rand_delete	4,194,304	3.00	333.3
libart	rand_insert	65,536	5.32	188.0
libart	rand_insert	262,144	5.51	181.4

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
libart	rand_insert	1,048,576	4.71	212.3
libart	rand_insert	4,194,304	3.41	293.7
libart	search_after_churn	65,536	14.80	67.6
libart	search_after_churn	262,144	9.33	107.1
libart	search_after_churn	1,048,576	6.66	150.3
libart	search_after_churn	4,194,304	5.85	170.9
libart	seq_insert	65,536	6.92	144.5
libart	seq_insert	262,144	6.96	143.7
libart	seq_insert	1,048,576	8.28	120.8
libart	seq_insert	4,194,304	8.79	113.7
libart	ycsb_a	65,536	5.48	182.6
libart	ycsb_a	262,144	6.55	152.7
libart	ycsb_a	1,048,576	7.22	138.5
libart	ycsb_a	4,194,304	7.76	128.9
libart	ycsb_b	65,536	12.85	77.8
libart	ycsb_b	262,144	9.60	104.2
libart	ycsb_b	1,048,576	6.35	157.4
libart	ycsb_b	4,194,304	4.91	203.8
matryoshka	mixed	65,536	0.08	12,882.6
matryoshka	mixed	262,144	0.07	14,596.7
matryoshka	mixed	1,048,576	0.08	11,874.3
matryoshka	mixed	4,194,304	0.08	11,952.5
matryoshka	rand_delete	65,536	0.08	12,023.7
matryoshka	rand_delete	262,144	0.08	12,374.2
matryoshka	rand_delete	1,048,576	0.08	12,494.9
matryoshka	rand_delete	4,194,304	0.08	12,501.5
matryoshka	rand_insert	65,536	0.08	12,515.2
matryoshka	rand_insert	262,144	0.08	12,469.8
matryoshka	rand_insert	1,048,576	0.08	12,380.3
matryoshka	rand_insert	4,194,304	0.08	12,523.4
matryoshka	search_after_churn	65,536	0.46	2,166.0
matryoshka	search_after_churn	262,144	0.43	2,331.3
matryoshka	search_after_churn	1,048,576	0.41	2,429.6
matryoshka	search_after_churn	4,194,304	0.38	2,664.3
matryoshka	seq_insert	65,536	0.09	10,843.0
matryoshka	seq_insert	262,144	0.09	10,901.5
matryoshka	seq_insert	1,048,576	0.08	11,845.8
matryoshka	seq_insert	4,194,304	0.08	11,900.7
matryoshka	ycsb_a	65,536	0.08	12,027.5
matryoshka	ycsb_a	262,144	0.09	11,105.7
matryoshka	ycsb_a	1,048,576	0.09	11,104.0
matryoshka	ycsb_a	4,194,304	0.09	11,457.4
matryoshka	ycsb_b	65,536	0.15	6,813.4
matryoshka	ycsb_b	262,144	0.14	7,192.9
matryoshka	ycsb_b	1,048,576	0.14	7,316.7
matryoshka	ycsb_b	4,194,304	0.13	7,444.8

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
std_set	mixed	65,536	2.52	397.1
std_set	mixed	262,144	1.72	580.0
std_set	mixed	1,048,576	1.00	1,000.2
std_set	mixed	4,194,304	0.71	1,403.0
std_set	rand_delete	65,536	2.11	474.6
std_set	rand_delete	262,144	1.04	958.7
std_set	rand_delete	1,048,576	0.55	1,803.6
std_set	rand_delete	4,194,304	0.42	2,396.8
std_set	rand_insert	65,536	2.35	425.2
std_set	rand_insert	262,144	1.79	559.7
std_set	rand_insert	1,048,576	0.88	1,130.5
std_set	rand_insert	4,194,304	0.47	2,142.4
std_set	search_after_churn	65,536	2.34	428.2
std_set	search_after_churn	262,144	1.18	850.9
std_set	search_after_churn	1,048,576	0.61	1,636.0
std_set	search_after_churn	4,194,304	0.38	2,612.4
std_set	seq_insert	65,536	3.76	265.6
std_set	seq_insert	262,144	2.52	396.7
std_set	seq_insert	1,048,576	2.17	460.0
std_set	seq_insert	4,194,304	1.72	582.5
std_set	ycsb_a	65,536	3.75	266.7
std_set	ycsb_a	262,144	2.18	459.0
std_set	ycsb_a	1,048,576	1.98	504.6
std_set	ycsb_a	4,194,304	1.55	645.6
std_set	ycsb_b	65,536	2.20	453.8
std_set	ycsb_b	262,144	1.26	796.2
std_set	ycsb_b	1,048,576	0.61	1,636.3
std_set	ycsb_b	4,194,304	0.41	2,432.4
tlx_btree	mixed	65,536	4.45	224.7
tlx_btree	mixed	262,144	4.06	246.4
tlx_btree	mixed	1,048,576	3.09	323.9
tlx_btree	mixed	4,194,304	2.17	459.9
tlx_btree	rand_delete	65,536	3.34	299.6
tlx_btree	rand_delete	262,144	2.73	366.6
tlx_btree	rand_delete	1,048,576	2.14	466.4
tlx_btree	rand_delete	4,194,304	1.46	687.0
tlx_btree	rand_insert	65,536	3.84	260.3
tlx_btree	rand_insert	262,144	3.26	307.0
tlx_btree	rand_insert	1,048,576	2.22	449.6
tlx_btree	rand_insert	4,194,304	1.70	586.8
tlx_btree	search_after_churn	65,536	3.68	271.4
tlx_btree	search_after_churn	262,144	2.82	354.6
tlx_btree	search_after_churn	1,048,576	2.03	491.4
tlx_btree	search_after_churn	4,194,304	1.39	717.6
tlx_btree	seq_insert	65,536	9.92	100.8
tlx_btree	seq_insert	262,144	10.50	95.2

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
tlx_btree	seq_insert	1,048,576	9.34	107.1
tlx_btree	seq_insert	4,194,304	8.83	113.2
tlx_btree	ycsb_a	65,536	9.24	108.2
tlx_btree	ycsb_a	262,144	8.32	120.1
tlx_btree	ycsb_a	1,048,576	8.01	124.9
tlx_btree	ycsb_a	4,194,304	6.64	150.5
tlx_btree	ycsb_b	65,536	3.19	313.7
tlx_btree	ycsb_b	262,144	2.92	342.3
tlx_btree	ycsb_b	1,048,576	2.02	495.9
tlx_btree	ycsb_b	4,194,304	1.36	737.5

10 Analysis and Diagnosis

10.1 The O(B) Leaf Rebuild Cost

The dominant cost is the *full leaf rebuild*. Unlike a sorted-array B-tree leaf (where insert is a `memmove` of $\sim B/2$ keys), matryoshka must:

1. **Extract** all ≤ 511 keys from the FAST layout via `mt_leaf_extract_sorted` (iterates 512 slots, dereferences `sorted_rank[]`).
2. **Insert/remove** the target key (binary search + `memmove`).
3. **Rebuild** the full FAST layout via `mt_leaf_build`: BFS tree, in-order map, recursive hierarchical blocked layout.

This is $\sim 3 \times 511 \approx 1{,}533$ key touches per insert vs. ∼255 for a sorted-array leaf. `mt_leaf_build` consumes N/A% of CPU during random insert at N=1,048,576.

Matryoshka achieves 0.08 Mop/s on random insert (N=1,048,576), vs. 0.88 Mop/s (`std::set`) and 4.71 Mop/s (fastest B-tree competitor).

10.2 SIMD Blocking: Search Benefit, Zero Modification Benefit

The FAST layout delivers ∼4.5× fewer comparisons per search than linear scan, using `_mm_cmpgt_epi32` + `_mm_movemask_ps` to resolve 3 keys per step. This yields 0.41 Mop/s on `search_after_churn` (N=1,048,576), fastest among all libraries.

However, SIMD provides *zero benefit* during modification: the insert/delete path unconditionally extracts and rebuilds without performing any SIMD search within the modified leaf.

10.3 Arena Allocator and TLB Effects

The arena allocator places leaves in superpage-aligned regions:

- **Reduced dTLB misses**: one 2 MiB superpage covers 512 leaf pages. dTLB rate: 0.1/1,000 ops vs. 42.1 for `std::set`.
- **Prefetch**: contiguous pages benefit sequential scans.

During insert/delete, TLB effects are secondary to the $O(B)$ rebuild cost.

10.4 Where std::set Falls Behind

`std::set` (red-black tree) suffers from pointer chasing (one cache miss per level), poor spatial locality, and high per-node overhead (40–48 B/key vs. 4 B in matryoshka). Despite this, its

$O(\log N)$ insert with a constant-factor pointer update often beats matryoshka's $O(B)$ rebuild when B is large.

10.5 Where TLX and Abseil Compete

Both use sorted-array B-tree leaves with `memmove` insert ($B \approx 64$–256). They stay within 9% of each other and consistently outperform matryoshka on modification because their per-insert constant factor is far lower. Neither uses SIMD for in-leaf search.

10.6 ART's Radix Approach

ART performs $O(\text{key\_length})$ operations independent of N. For 4-byte keys it traverses ≤ 4 levels with compact node arrays (4–256 entries). Insert and delete are cache-efficient, but ART lacks native predecessor search, so `search_after_churn` uses point lookups.

10.7 Overall Diagnosis

Matryoshka trades modification throughput for search throughput. The FAST layout minimises cache misses and branch mispredictions during search, but the $O(B)$ extract-and-rebuild per insert/delete makes it 43× slower on insert-heavy and 38× slower on delete-heavy workloads than the best B-tree competitor. This trade-off is inherent: the hierarchical blocking that accelerates search makes in-place modification impossible.

11 Improvement Recommendations

11.1 Incremental Leaf Update

Avoid full rebuild for single-key changes. Compute the incremental BFS change and rewrite only the $O(\log B)$ affected SIMD/cache-line blocks and their `sorted_rank[]` entries. Expected 3–5× speedup for single-key ops.

11.2 Batch Insert API

Provide `matryoshka_insert_batch(tree, keys, k)`: sort incoming keys, merge into each affected leaf's sorted array, rebuild once per leaf. Amortises cost from $k \times O(B)$ to $O(B + k \log k)$. Batch sizes of 32–64 reduce per-key cost by 20–50×.

11.3 Write-Optimised Leaf Variant

Dual-mode leaf:

- **Sorted-array mode** for small/frequently modified leaves: `memmove` insert, SIMD binary search (no blocking).
- **FAST mode** for large/read-heavy leaves: current hierarchical layout, activated after T searches without modification or at a size threshold.

Transition uses existing `mt_leaf_build` / `mt_leaf_extract_sorted`.

11.4 Superpage Hierarchy for Large Datasets

Use 2 MiB superpage leaves (`mt_hierarchy_init_superpage`) holding ∼131,071 keys (depth 17). Reduces leaf-level TLB entries by 512×. Requires batch insert to amortise the larger per-leaf rebuild.

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.