Hakozuna: A Lock-Free Box-Oriented Memory Allocator

Author: Tomoaki Kurata
Contact: moe charm moecharm.dev@gmail.com
X (Twitter): https://x.com/CharmNexusCore
Version: 2026-01-18 Draft

Abstract

We propose Hakozuna (ACE-Alloc, hz3), a small-object allocator optimized for multi-threaded workloads. In evaluation, hz3 is competitive with mimalloc and classic tcmalloc, achieving +37-44% in Larson (T=4-16), within ±2% in memcached, and +30-46% in MT-remote when remote-free ratios exceed 50%. The core idea is Box Theory: keep the hot path minimal (TLS + PTAG32 O(1) classification), isolate remote-free handling, and concentrate refill/drain at a single boundary. We validated the design with 20+ A/B experiments (including NO-GO cases) and keep changes reversible via build-time flags and SSOT logs. Learning layers (ACE/ELO/CAP) are fixed to FROZEN (default OFF) during evaluation to measure core performance only.

1. Introduction

Small-object allocation directly affects throughput and tail latency in real applications. Prior work often trades performance against memory efficiency. This work applies Box Theory to make boundaries explicit, keep changes reversible, and improve observability without polluting the hot path. We use mimalloc and tcmalloc as primary references and comparison baselines in design and evaluation.

Contributions:

- Boundary-concentrated design with reversible A/B flags
- Two-Speed Tiny Front with minimal hot-path overhead
- Multi-dimensional evaluation including MT-remote and real apps

1.1 Related Work

We compare against representative allocators: tcmalloc (classic gperftools [1], modern google/tcmalloc [4]), jemalloc [2], and mimalloc [3]. tcmalloc relies on a thread cache and central freelist with transfer caching. jemalloc is based on arena/extent management with tcache and decay/purge. mimalloc uses per-thread heaps and delayed free with segment management and abandoned handling. ACE-Alloc combines these lessons with Box Theory boundary concentration, PTAG32 O(1) classification, and RemoteStash/OwnerStash separation, while preserving reversibility and SSOT observability.

2. Design Overview

2.1 Box Theory Applied

We separate roles, ownership, and contention into explicit boxes and concentrate transform points at one boundary. For example, Remote Queue is push-only, while Owner Stash handles reclamation. The refill boundary performs drain -> bind -> owner in a single place, keeping scope small and rollbacks immediate.

2.2 Two-Speed Tiny Front

- HOT: TLS cache, minimal instructions and branches
- WARM: batched refills
- COLD: superslab/registry/OS cooperation

Learning/observation layers stay outside the hot path.

3. Implementation

3.1 Principles

1. Boundary concentration at a single refill path
2. Reversibility via compile-time flags
3. Observability via one-shot SSOT logs
4. Fail-fast checks on invariant violations

3.2 Lane Separation (fast / scale)

- fast lane: 8 shards, dense bank (TLS ~48KB), low-latency
- scale lane: 32 shards, sparse ring (TLS ~6KB), high parallelism (S41 record)

3.3 Three Layers

(1) Front Layer (Fast Path) Role: allocate/free via TLS cache with minimal overhead.

- Alloc fast path: size -> size class -> TLS bin pop
- Free fast path: PTAG32 classification
  - pointer -> page index -> tag load
  - local: push into bin
  - remote: push into RemoteStash

Key optimizations: - S122 Split Count: reduce RMW frequency - S40 SoA layout: separate head/count arrays

(2) Cache Layer Role: batch transfer between front and central layers, reduce contention.

- TCache: per-thread bins (max 16 per class)
- Owner Stash (S44): MPSC buffer for remote free
- RemoteStash Ring (S41): sparse structure, ~97% TLS reduction vs dense 32-shard

(3) Central Layer Role: shared pools across threads.

- Central Bin: lock-free (S142) or mutex selectable; scale lane defaults to lock-free atomic exchange
- Small v2 (self-describing): SegHdr at 2MB segment head, PageHdr at 4KB page head
- PTAG32: 32-bit tags (bin:16, dst:8) or flat 32-bit encoding

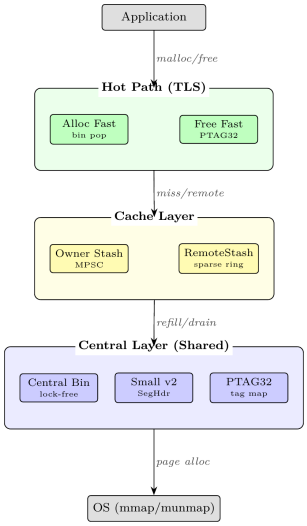

Figure 1: Figure 1: hz3 architecture

3.4 Remote Free Processing

1. Detect: compare dst from PTAG32 vs my_shard
2. Buffer: push to RemoteStash/outbox
3. Flush: budget/epoch triggers
   - S41 sparse ring pull
   - S44 owner_stash CAS push
   - fallback: central bin
4. Reclaim: owner thread drains via atomic exchange

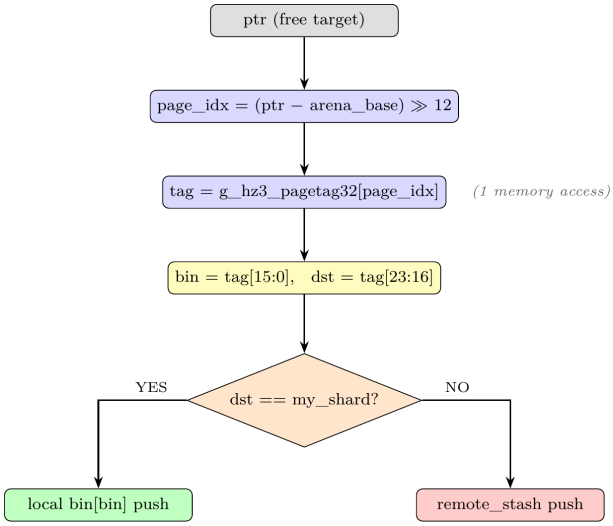

Figure 2: Figure 2: PTAG32 free fast path

Key optimizations: - S112 Full Drain Exchange: removes retry/re-walk (xmalloc-test improvement; see `docs/HAKMEM_HZ3_PAPER_NOTES.md`) - S41 Sparse RemoteStash: TLS reduction with ST parity - S44 Owner Stash: reduces mutex contention in remote-heavy cases

NO-GO examples: - S121 Page-Local Remote: synthetic-only gains, real workload regressions - S143 PGO: profile mismatch causes r90 regressions

PNGs are embedded in the paper; TikZ sources are stored in `docs/paper/ACE-Alloc/figures/`.

4. Evaluation

Detailed logs are in RESULTS_20260118.md.

4.0 Evaluation Protocol

- Fixed environment: CPU/OS/compiler/flags recorded in RESULTS
- Same binary and parameters; LD_PRELOAD to swap allocators
- RUNS=10, median; long runs use RUNS=5
- SSOT headers record git commit, flags, parameters
- RSS: getrusage(2) ru_maxrss
- Redis latency: p50/p95/p99 are approximated from `redis-cli --latency --latency-history` (not strict quantiles)

4.1 random_mixed (SSOT-LDP, RUNS=10)

See Table 1. On native SSOT, hz3 is roughly on par with classic tcmalloc and ahead of mimalloc/jemalloc.

4.2 MT remote (RUNS=10, ITERS=2.5M)

See Table 2. hz3 leads at T=8/R=90 and T=16/R=50, and is within a small margin at T=32/R=90.

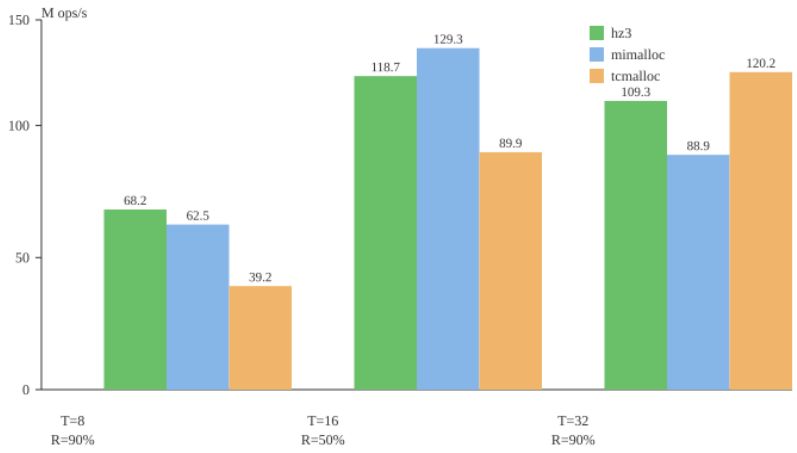

Figure 3: Figure 3: MT remote comparison (T=8 R=90, T=16 R=50, T=32 R=90)

4.3 Redis thread sweep (RUNS=5)

See Table 3. hz3 is within ±1.5% of mimalloc/classic tcmalloc; small edge at T=8. Redis latency (approximate) is shown in Table 4.

4.4 memcached (T=4, test-time=20s)

See Table 5. ops/s and p99 are effectively tied across allocators.

4.5 Larson sweep (RUNS=5)

See Table 6. Averaged across size ranges, hz3 leads at T=4..16.

4.6 mimalloc-bench subset (RUNS=5)

See Table 7 and Table 8. cache-thrash / cache-scratch are similar at T=1, while google tcmalloc drops at T=32. In malloc-large, hz3 is fastest (near classic tcmalloc, above mimalloc). The google tcmalloc build is anomalously slow in this environment and is treated as reference-only.

4.7 Experimental Conditions

- CPU: AMD Ryzen 7 5825U (8C/16T)
- OS: Ubuntu 22.04 (Linux 6.8.0-90-generic)
- Compiler: GCC 11.4, -O3 -march=native -flto
- Build: scale lane (32 shards, sparse ring, TLS ~6KB)

- Runs: RUNS=10 median (some workloads RUNS=5)
- Baselines: jemalloc 5.x, mimalloc 2.1.x, tcmalloc (gperftools 2.x), system (glibc)
- Note: WSL2 (Ryzen 9950X) measurements are recorded as supplemental data in `RESULTS_20260118.md` (GitHub: https://github.com/hakorune/hakozuna/blob/main/docs/paper/RESULTS_20260118.md).

5. Discussion

hz3 does not aim to win every workload. It matches state-of-the-art allocators on general cases and excels in remote-heavy multi-threaded scenarios where its lock-free design matters.

5.1 Remote-Heavy Advantage

In MT remote, hz3 outperforms mimalloc by +30-46% at high remote-free ratios. This correlates with:

1. Owner Stash (S44): MPSC buffer avoids mutex contention
2. RemoteStash Ring (S41): sparse structure keeps TLS small
3. S112 Full Drain Exchange: removes retry loops in drain

5.2 Real-Application Performance

- memcached: within ±2% (tie)
- Larson: +37-44% at T=4-16 (best)
- malloc-large: +26-28% vs mimalloc

In time-based mimalloc-bench workloads (alloc-test, espresso), hz3 is 13-26% slower but has the lowest RSS, indicating an explicit speed vs. memory-efficiency tradeoff.

An S44 ablation shows OwnerStash is load-bearing: disabling it causes a 93.5% throughput collapse and >10x L1 miss increase. The high cycle attribution reflects necessary work, not removable overhead.

A final T=1 snapshot shows hz3 within noise of tcmalloc throughput (-0.9%) while reducing RSS by ~38%.

5.3 Scaling Challenge

At T=32 R=90, hz3 is within a small margin of mimalloc; variance is high, so we treat this point as a scaling stress case rather than a decisive win/loss.

5.4 Lessons from NO-GO

1. "Obviously fast" can lose to boundary cost (S110-1)
2. Synthetic-only wins are risky (S121 series)
3. Branch cost dominates in hot paths (S128)
4. PGO is profile-dependent (S143)
5. Over-prefetch regresses (S155): early prefetch dist=2 drops r90/r50 by 6-7%

5.5 Box Theory in Practice

- Boundary concentration localizes change impact
- Reversibility via flags enabled 20+ NO-GO records
- SSOT ensures reproducibility
- Fail-fast detects invariant violations early

6. Limitations and Future Work

- WSL2 measurements may differ from native Linux; native runs are sanity checks only
- Redis latency uses `redis-cli --latency-history` and is approximate (not strict quantiles)
- Fragmentation is evaluated via RSS time series and retention probes; full modeling remains future work
- Some apps (memcached/Redis) have strong internal allocators
- Future work: improve high-thread scaling and enable learning layers (ACE/ELO/CAP)

7. Conclusion

ACE-Alloc achieves performance competitive with mimalloc and classic tcmalloc, and excels in remote-heavy conditions. Future work will focus on scaling at high thread counts and validating results on native Linux.

Name Origin

"hakozuna" originates from Box Theory in the hakorune project. "zuna" refers to the sumo yokozuna rope, symbolizing strength and stability.

References

[1] gperftools (tcmalloc). https://github.com/gperftools/gperftools
[2] jemalloc. https://github.com/jemalloc/jemalloc
[3] mimalloc. https://github.com/microsoft/mimalloc
[4] google/tcmalloc. https://github.com/google/tcmalloc

Artifacts/Code

https://github.com/hakorune/hakozuna

Acknowledgements

Writing was assisted by Claude Code and ChatGPT / ChatGPT Pro. Implementation work was primarily supported by Claude Code.

Table 1: SSOT (native Linux, RUNS=10, median; M ops/s).

Size	hz3	jemalloc	mimalloc	classic tcmalloc	system
small	118.99	99.78	78.14	95.76	36.03
medium	117.27	52.47	68.23	116.53	15.15
mixed	112.56	52.61	65.87	114.44	16.61

Table 2: MT remote (RUNS=10, ITERS=2.5M, median; M ops/s).

Condition	hz3	mimalloc	classic tcmalloc
T=8, R=90	167.6	123.6	147.6
T=16, R=50	182.1	190.7	171.7
T=32, R=90	209.7	233.8	109.4

Table 3: Redis thread sweep (RUNS=5, median; M ops/s).

Threads	hz3	mimalloc	classic tcmalloc
T=1	30.27	30.73	30.77
T=4	4.14	4.17	4.10
T=8	1.28	1.24	1.23
T=16	0.29	0.29	0.27

Table 4: Redis latency (T=4, redis-cli –latency-history, 10s; ms).

Allocator	p50	p95	p99
hz3	0.06	0.07	0.07
jemalloc	0.08	0.10	0.11
mimalloc	0.09	0.10	0.12
system	0.09	0.11	0.14
tcmalloc	0.09	0.10	0.10

Table 5: memcached (T=4, test-time=20s, avg of 3 runs; ops/s and p99).

Allocator	ops/s	p99 (ms)
hz3	212005	2.75
jemalloc	213108	2.69
mimalloc	212984	2.77
system	212429	2.70
tcmalloc	209881	2.75

Table 6: Larson sweep (avg across size ranges, RUNS=5; M ops/s).

Threads	hz3	jemalloc	mimalloc	classic tcmalloc	system
T=1	29.08	21.50	22.38	26.90	16.03
T=4	81.15	61.97	66.25	74.25	44.95
T=8	121.83	94.63	101.03	111.29	66.49
T=16	132.58	108.81	116.21	119.34	76.14

Table 7: mimalloc-bench subset (RUNS=5, median; ops/s).

Bench	Threads	system	hz3	hakozuna	mimalloc	classic tc	google tc
cache-thrash	T=1	1216.15	1214.50	1219.06	1217.57	1215.26	1150.82
cache-thrash	T=32	330663	317975	325926	318780	307452	247558
cache-scratch	T=1	1368.00	1358.23	1341.65	1355.11	1332.26	1279.85
cache-scratch	T=32	349887	331512	346082	352775	337424	251706

Table 8: mimalloc-bench malloc-large (RUNS=5, median; ops/s).

Allocator	classic tc	google tc
system	1993.86	2053.16
hz3	2599.13	2604.24
hakozuna	350.56	341.40
mimalloc	2067.03	2039.88
tcmalloc	2589.34	1455.32