

# Hakozuna: ロックフリーな Box 指向メモリアロケータ

著者: 倉田 智朗 (TOMOAKI KURATA)

連絡先: moe charm moecharm.dev@gmail.com

X (Twitter): <https://x.com/CharmNexusCore> 版: 2026-01-18 Draft

## 概要 (Abstract)

本稿では、mimalloc と同等以上、classic tcmalloc とほぼ同等の性能を示す小型オブジェクト向けアロケータ「Hakozuna (ACE-Alloc, hz3)」を提案する。評価では、Larson で mimalloc 比+37~44% (T=4-16)、memcached では±2%以内で拮抗、MT remote で remote-free 比率 50%以上の条件において mimalloc 比+30~46%の優位を確認した。設計の核心は Box Theory に基づく境界集約であり、ホットパス (TLS) と制御層 (refill/drain) を分離し、変換点を 1 箇所限定する。20 以上の A/B 実験 (NO-GO 含む) で検証し、最適化はコンパイル時フラグで切り戻し可能とした。本稿の評価は ACE/ELO/CAP 学習層を FROZEN (既定 OFF) に固定し、hz3 コアの性能を測定した。

## 1. はじめに

小型オブジェクト割り当ては、実アプリのスループット・レイテンシに直結する。従来は「速さ」と「メモリ効率」のトレードオフに留まることが多かったが、本研究は Box Theory による設計分割で、可観測性と切り戻し可能性を維持したまま性能を積み上げる。設計にあたっては mimalloc と tcmalloc を主要な参考・比較対象とし、評価で性能と安定性を検証した。

本稿の貢献は以下である。

- Box Theory に基づく境界集約 (変換点 1 箇所) と“戻せる” A/B 設計
- Two-Speed Tiny Front を中心とした低オーバーヘッド設計
- 実アプリおよび MT remote を含む多面的評価

### 1.1 関連研究

既存の代表的アロケータとして、tcmalloc (classic: gperftools [1], modern: google/tcmalloc [4])、jemalloc [2]、mimalloc [3] を比較対象とした。tcmalloc は thread cache と central freelist を軸にし、transfer cache により スレッド間の負荷移送を行う。jemalloc は arena/extent を基盤に tcache と decay/purge でメモリ管理を行う。mimalloc は per-thread heap と delayed free を採用し、segment 管理と abandoned 処理で競合を抑える。hz3 (ACE-Alloc) はこれらの設計に学びつつ、Box Theory による境界集約、PTAG32 による即時分類、RemoteStash/OwnerStash による remote-free の分離、A/B 切替と SSOT による可観測性を組み合わせる点で差別化する。

## 2. 設計の全体像

### 2.1 Box Theory の適用

本研究では「役割・責務・所有」が交わる場所を箱として切り分け、変換点を 1 箇所に集約する。箱は“人間の境界”であり、所有や競合制御を箱の内側に閉じ込める。たとえば Remote Queue は push のみに限定し、Owner Stash は回収のみを担当する。採用境界 (refill 経路) でのみ drain → bind → owner をまとめて行うことで、影響範囲と切り戻しを明確にする。

## 2.2 Two-Speed Tiny Front

- HOT: TLS キャッシュを中心に、最小命令・最小分岐で処理
- WARM: バッチ補充 (refill) でリクエストを吸収
- COLD: Superslab / Registry / OS 連携を担当

学習・観測層は上位箱に隔離し、HOT パスへ影響を与えない。

## 3. 実装

### 3.1 設計方針

hz3 は、Box Theory (箱理論) に基づき、以下の 4 原則を徹底する：

1. **境界集約**: ホットパスと制御層の境界を最小化し、変換点を 1 箇所 (refill 経路) に集約
2. **可逆性**: コンパイル時フラグによる A/B 切り替えで、最適化を戻せる設計
3. **可観測性**: atexit 時の one-shot ログ (SSOT) により、再現性を確保
4. **Fail-Fast**: 不正状態を境界で検出し、早期に異常終了

### 3.2 レーン分離 (fast / scale)

hz3 は用途に応じて 2 つのビルド構成を提供する：

- **fast lane**: 8 シャード、dense bank 構造 (TLS ~48KB)、低レイテンシ重視
- **scale lane**: 32 シャード、sparse ring 構造 (TLS ~6KB)、高並列度対応 (S41 実装記録)

### 3.3 3 層構造

(1) フロント層 (Fast Path) 役割: 最小命令・最小分岐で TLS キャッシュから割り当て・解放

- **Alloc Fast Path**: サイズ→サイズクラス変換→TLS ローカル bin から pop
- **Free Fast Path**: PageTagMap (PTAG32) による高速分類
  - ポインタ→ページインデックス変換 (1 算術演算)
  - tag 読み込み (1 メモリアクセス) で (bin, dst) を取得
  - ローカル: bin へ push
  - リモート: RemoteStash へ遅延 push

最適化技術: - **S122 Split Count**: bin カウント更新を 16 分周化、RMW 削減 - **S40 SoA Layout**: head 配列と count 配列を分離、キャッシュライン効率向上

(2) キャッシュ層 (Cache Management) 役割: フロント層と central 層の間でバッチ転送、ロック競合削減

- **TCache (Thread Cache)**: 各スレッド専用の Hz3TCache を TLS で保持、サイズクラス毎に最大 16 個キャッシュ
- **Owner Stash (S44)**: remote free 時の中間バッファ (MPSC: Multiple Producer Single Consumer)
  - 他スレッドからの remote free を CAS で追加 (ロックフリー)
  - 自スレッドの alloc miss 時、central アクセス前に一括取得
  - 効果: mutex 競合回避、remote-heavy 条件でスループット向上

- **RemoteStash Ring (S41, scale lane)**: sparse 構造による TLS 削減 (dense 32-shard 比 約 97%削減、S41 実装記録)

(3) **Central 層 (Shared State)** 役割: スレッド間で共有されるオブジェクトプールを管理

- **Central Bin**: lock-free (S142) または mutex 保護の freelist を選択可能。scale lane 既定は lock-free で atomic exchange により一括取得/push
- **Small v2 (self-describing)**: segmap 不要の自己記述型設計
  - SegHdr: 各 2MB セグメントの先頭に magic/owner/page\_meta を配置
  - PageHdr: 各 4KB ページの先頭に magic/sc/owner を配置
- **PageTagMap (PTAG32)**: ポインタからサイズクラスと owner を高速に逆引き
  - 32bit タグ (bin: 16bit, dst: 8bit) または flat 32bit で bin/dst を提供
  - 更新: page 割り当て時に atomic store

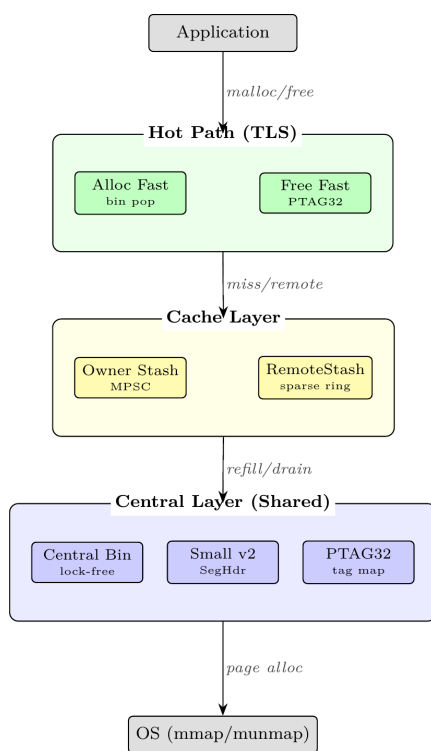


Figure 1: 図 1: hz3 全体アーキテクチャ

### 3.3.1 境界 (変換点) のまとめ

層間の変換点は refill 経路に集約し、境界を越える副作用を禁止する。

- **Hot → Cache**: miss / remote のみ。共有状態には触れない。
- **Cache → Central**: refill / drain のみ。publish や owner の変更はここに集約。
- **Central → OS**: segment / page の取得・返却のみ。Hot 側へ状態を持ち込まない。

### 3.4 Remote Free 処理

remote free は以下の段階を経る：

1. 検出: PTAG32 から取得した dst と my\_shard を比較
2. 一時保存: RemoteStash または outbox へ push (thread-local 操作)
3. フラッシュ: バジエツト超過時または epoch 時
  - S41: sparse ring から (dst, bin, ptr) を取り出し
  - S44: owner\_stash へ CAS push (ロックフリー)
  - fallback: central bin へ mutex 経由で push
4. 回収: owner thread の alloc miss 時、owner\_stash から atomic exchange で一括 pop

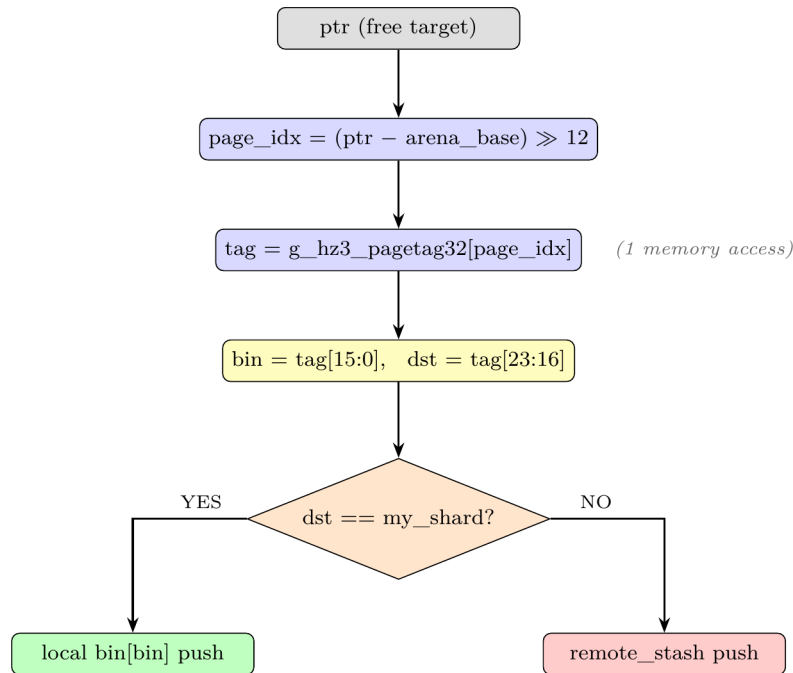


Figure 2: 図 2: PTAG32 Free Fast Path

**主要最適化:** - **S112 Full Drain Exchange:** bounded drain の retry+re-walk を atomic\_exchange で除去 (xmalloc-test で改善。詳細は docs/HAKMEM\_HZ3\_PAPER\_NOTES.md 参照) - **S41 Sparse RemoteStash:** TLS 約 97%削減、ST 性能維持 - **S44 Owner Stash:** remote-heavy 条件で mutex 競合削減

**NO-GO 事例** (採用されなかった最適化)： - **S121 シリーズ Page-Local Remote:** synthetic (xmalloc-test) 依存で実ワークロード退行 - **S143 PGO:** プロファイル不一致で r90 退行

### 3.5 学習層の位置づけ (ACE / ELO)

学習層は「ポリシー箱」に閉じ込め、Hot Path に影響を与えない。

- **FROZEN が既定** (学習 OFF)
- **Hot Path は “現在のポリシー値” のみ参照**
- 学習の ON/OFF は ENV で切替可能 (戻せる設計)

本文には PNG を埋め込み、TikZ 原稿は docs/paper/ACE-Alloc/figures/ に保存した。

## 4. 評価

評価の詳細ログは RESULTS\_20260118.md に集約する。本稿では主要結果のみを示す。

### 4.0 評価プロトコル

- **環境固定:** CPU/OS/コンパイラ/フラグは RESULTS\_20260118.md に記録したものを使用 (GitHub: [https://github.com/hakorune/hakozuna/blob/main/docs/paper/RESULTS\\_20260118.md](https://github.com/hakorune/hakozuna/blob/main/docs/paper/RESULTS_20260118.md))。
- **同一条件の比較:** ベンチ実行は同一バイナリ・同一パラメータで、LD\_PRELOAD で allocator を差し替える。
- **統計処理:** RUNS=10 を基本とし中央値を採用 (長尺系は RUNS=5)。
- **ログの一貫性:** SSOT ヘッダに git commit / build flags / 実行パラメータを記録。
- **RSS:** getrusage(2) の ru\_maxrss で最大 RSS を記録。
- **評価表 (LaTeX):** docs/paper/ACE-Alloc/tables.tex にまとめた。
- **Redis レイテンシ:** redis-cli --latency --latency-history の履歴値から p50/p95/p99 を近似 (厳密な分位ではない)。

#### 4.1 random\_mixed (SSOT-LDP, RUNS=10)

表 1 を参照。native SSOT では hz3 は classic tcmalloc とほぼ同等で、mimalloc/jemalloc を上回る。

#### 4.2 MT remote (RUNS=10, ITTERS=2.5M)

表 2 を参照。T=8/R=90 と T=16/R=50 で hz3 が優位で、T=32/R=90 は mimalloc と僅差。

#### 4.3 Redis thread sweep (RUNS=5)

表 3 を参照。全体的に  $\pm 1.5\%$  程度で拮抗し、T=8 で hz3 がわずかに上回る。Redis のレイテンシ近似は表 4 に示す (近似値である点に注意)。

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

表 5 を参照。ops/s と p99 は allocator 間でほぼ同等。

#### 4.5 Larson sweep (RUNS=5)

表 6 を参照。サイズ範囲の平均では T=4~16 で hz3 が最速。

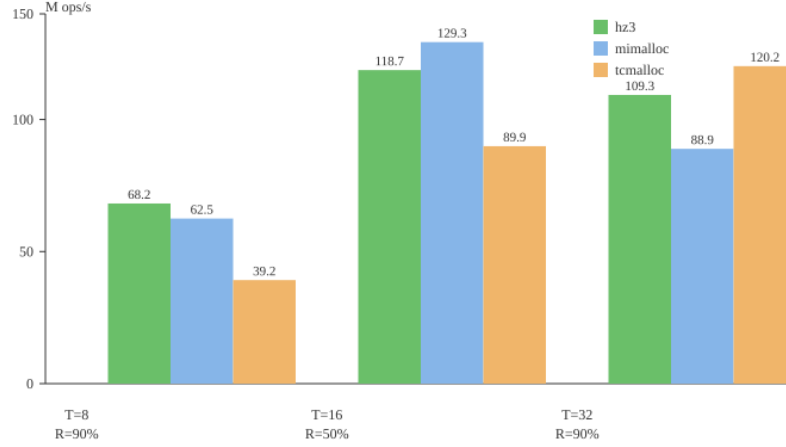


Figure 3: 図 3: MT remote 比較 (T=8 R=90, T=16 R=50, T=32 R=90)

#### 4.6 mimalloc-bench subset (RUNS=5)

表 7 と表 8 を参照。cache-thrash / cache-scratch は T=1 で概ね同等だが、T=32 で google tcmalloc が大きく低下。malloc-large は hz3 が最速で、classic と同等、mimalloc より高スループット。google tcmalloc は本環境で極端に低速だったため、参考値として扱う。

#### 4.7 実験条件

- CPU: AMD Ryzen 7 5825U (8C/16T)
- OS: Ubuntu 22.04 (Linux 6.8.0-90-generic)
- コンパイラ: GCC 11.4、最適化 -O3 -march=native -flto
- ビルド構成: scale lane (32 shards, sparse ring, TLS 6KB)
- ラン数: RUNS=10 (中央値採用)、一部ベンチは RUNS=5
- 比較対象: jemalloc 5.x、mimalloc 2.1.x、tcmalloc (gperftools 2.x)、system (glibc)
- 補足: WSL2 (Ryzen 9950X) の測定は参考値として RESULTS\_20260118.md に記録 (GitHub: [https://github.com/hakorune/hakozuna/blob/main/docs/paper/RESULTS\\_20260118.md](https://github.com/hakorune/hakozuna/blob/main/docs/paper/RESULTS_20260118.md))。

再現手順:

```
# ビルド (scale lane)
make -C hakozuna/hz3 clean all_ldpreload_scale

# SSOT 実行 (small/medium/mixed 一括)
RUNS=10 ITERS=20000000 WS=400 \
./hakozuna/hz3/scripts/run_bench_hz3_ssot.sh

# A/B テスト (フラグ差分のみ)
make -C hakozuna/hz3 clean all_ldpreload_scale \
  HZ3_LDPRELOAD_DEFS_EXTRA='-DHZ3_S52_BESTFIT_RANGE=2'
```

ログは /tmp/hz3\_ssot\_YYYYMMDD\_HHMMSS/ に保存され、以下を含む： - [BENCH-HEADER]: git commit、ビルドフラグ、実行パラメータ - 各 run の中央値・最小・最大 - atexit 時の SSoT 統計 (alloc/free 回数、refill/drain 回数など)

## 5. 考察

### 5.1 Remote-Heavy 条件での優位性

MT remote (T=8 R=90, T=16 R=50) で hz3 が mimalloc に対し +46%、+30% の優位を示した。これは以下の設計が効いている：

1. **Owner Stash (S44)**: remote free 時の中間バッファが mutex 競合を回避
2. **RemoteStash Ring (S41)**: sparse 構造 (TLS 6KB) でキャッシュ効率を維持
3. **S112 Full Drain Exchange**: atomic exchange による一括回収で retry ループを除去

### 5.2 実アプリケーションでの性能

- **memcached**: T=1~16 で  $\pm 2\%$  以内 (拮抗)
- **Larson**: T=4-16 で +37~44% (最速)
- **malloc-large**: mimalloc 比 +26-28% (tcmalloc と同等)

これらの結果から、hz3 は実ワークロードでも同等以上の性能を示すことが確認できた。

一方、mimalloc-bench の time-based ベンチ (alloc-test, espresso) では、hz3 は 13-26%遅いが RSS は最小であり、速度とメモリ効率のトレードオフが明確に現れた。

T=1 の最終スナップショットでは、hz3 は tcmalloc と同等のスループット (-0.9%) かつ RSS を約 38%削減した。

S44 ablation では、OwnerStash を無効化するとスループットが 93.5% 崩壊し L1 ミスが 10 倍以上に増加した。S44 は削減対象ではなく、性能を支える“荷重支持”の最適化である。

### 5.3 スケーリング上の課題

T=32 R=90 は mimalloc と僅差であり、超高スレッド+超高 remote 比率でのスケーラビリティ課題は残る。今後の最適化候補：

- P95-C シリーズ (notify queue/mailbox shard) の継続
- central bin 競合のさらなる削減

### 5.4 NO-GO 事例から得られた教訓

20 以上の NO-GO 事例を記録し、以下の教訓を得た：

1. 「当たり前前に速い」は実測で否定される (S110-1 SegMath)：命中率 100%でも境界コストで負ける
2. **synthetic 依存は危険** (S121 シリーズ)：xmalloc-test 改善も他 MT 退行
3. **分岐予測と固定費の形が支配的** (S128 Defer Minrun)：hash probe の branch-miss 増で退行
4. **PGO はプロファイル依存** (S143)：r0/r50 改善も r90 退行
5. **過剰な prefetch は退行** (S155)：early prefetch dist=2 は r90/r50 で -6~7% 退行

### 5.5 Box Theory の実践効果

- **境界集約**: refill 経路への変換点集約により、最適化の影響範囲を明確化
- **可逆性**: コンパイル時フラグで A/B 切り替え、20 以上の NO-GO 事例を記録・再現可能
- **可観測性**: atexit one-shot ログ (SSOT) により、ビルド/実行条件を完全記録
- **Fail-Fast**: PTAG32 tag==0 検出、central bin list 破壊検出で UAF/double-free を早期発見

## 6. 限界と今後の課題

- WSL2 環境での計測であり、Linux native との乖離を完全に排除できない (native は sanity check に留まる)。
- Redis のレイテンシは `redis-cli --latency-history` に基づく近似値であり、厳密な分位ではない。
- 断片化は RSS 時系列と保持量の観測で評価しているが、厳密な fragmentation ratio のモデル化は今後の課題。
- 一部のベンチ (memcached/Redis) は内部アロケータが強く、差が縮む傾向がある。
- 今後の課題は、(1) 高スレッド・remote-heavy 条件でのスケーリング改善、(2) 学習層 (ACE/ELO/CAP) の有効化と安全な運用モードの確立である。

## 7. 結論

Box Theory に基づく hz3 は、mimalloc と同等以上の性能を複数ベンチで示し、remote-heavy 条件では明確な優位を示した。今後は高スレッド環境でのスケーリング改善と、native Linux 環境での再検証を進める。

## 付記 (名称の由来)

「hako~~z~~una」は、別プロジェクト「hako~~r~~une」の箱言語で生まれた Box Theory を起点にし、「zuna」は横綱の綱を意味する。設計哲学と目標 (強さ・安定) を象徴する名称として採用した。

## 参考文献

- [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>

## 謝辞

本稿の執筆は Claude Code と ChatGPT / ChatGPT Pro の支援を受けた。実装作業は主に Claude Code により補助された。

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	hakozone	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
hakozone	350.56	341.40
mimalloc	2067.03	2039.88
tcmalloc	2589.34	1455.32