

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: charmpic moecharm.dev@gmail.com

X (Twitter): <https://x.com/CharmNexusCore> 版: 2026-02-18 v3.0

概要 (Abstract)

本稿では、mimalloc / tcmalloc と競合可能な性能を示す小型オブジェクト向けアロケータ「Hakozuna (ACE-Alloc, hz3)」と、hz3 開発後に remote-heavy 条件へ特化して設計した「hz4 (message-passing profile)」を提案する。評価では Ubuntu (native) 上で random_mixed の SSOT を実施し、R=0 の local-only 条件では hz3 が最速 (359.6M ops/s)、かつ RSS 最小を確認した。remote-heavy 条件では hz4 (message-passing 版) が優位となり、用途別に勝ち筋が分かれることを示した。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。free routing は **PTAG32 を既定**とし、PTAG 方式はビルドで切替可能だが、本稿の評価は **PTAG32 既定**で実施した。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。本研究はさらに、**単独開発者と LLM の協働により、約 3 か月の短期間で state - of - the - art アロケータを上回る性能に到達した実証事例**として位置づけられる。

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。本研究の特徴は、LLM 協働による設計・実装・評価の高速反復により、短期間で SOTA 級性能に到達した点にある。また、開発の後半で「単一構成で全条件を取る」方針を改め、local-heavy 向けの hz3 と remote-heavy 向けの hz4 を分離した点を、実運用上の設計判断として重視する。

1.0 開発タイムライン (要約)

- **2025 - 10 - 22:** 初期ベンチ記録 (mid 2-32KB の比較)。
- **2025 - 10 - 26:** 重大な計測誤り (glibc malloc を測定) を発見し、ベンチ基盤を修正。真の性能は mimalloc 比 **8.8× 遅い**地点から再出発。
- **2026 - 01 - 24:** SSOT 計測で hz3/hz4 が tcmalloc/mimalloc と競合可能な水準へ到達。R=0 では hz3 が最速、R=90 では hz4 が競合し、用途別に勝ち筋が明確化。

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
- hz3 (local/redis 重視) と hz4 (remote/high-thread 重視) の dual-profile 方針
- 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、PTAG 方式の切替可能な routing、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**: routing は PTAG 方式の切替が可能

- PTAG32: 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
- 既定は PTAG32 (scale lane)

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

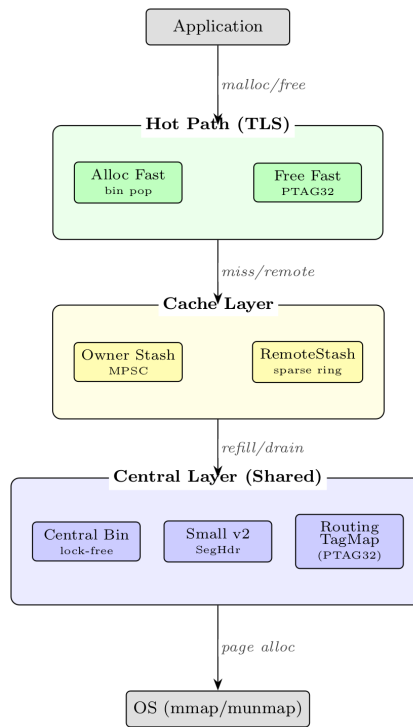


Figure 1: 図 1: hz3 全体アーキテクチャ

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

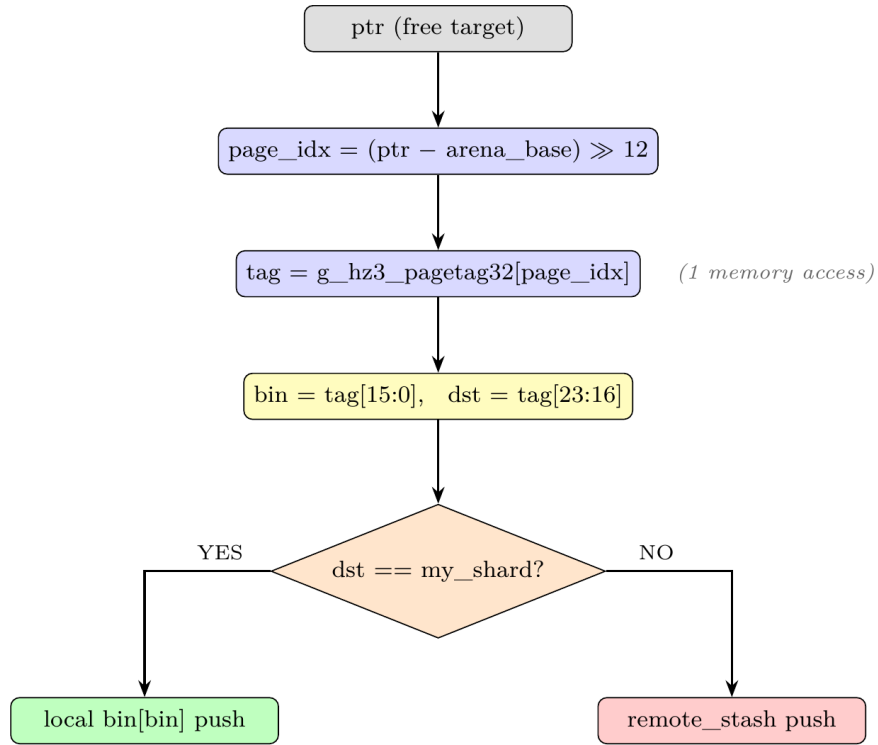


Figure 2: 図 2: PTAG32 Free Fast Path

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 docs/benchmarks/2026-02-18_PAPER_BENCH_RESULTS.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 docs/benchmarks/2026-02-18_PAPER_BENCH_RESULTS.md に記録したものを使用。
- **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。

4.1 SSOT (random_mixed, T=16/R=90, RUNS=10)

表 2 を参照。SSOT では hz4 が最速、hz3/tcmalloc/mimalloc が僅差で並ぶ構図となった。

4.2 T sweep (R=90, RUNS=3)

表 3 を参照。T=16 で hz4 が大きく伸び、メッセージパッシング設計のスケーリング効果が確認できる。

4.3 R sweep (T=16, RUNS=3)

表 4 を参照。R=0 では hz3 が最速、R=50 では tcmalloc が最速、R=90 では hz4 が最速となり、remote 比率に応じて勝ち筋が分かれる。

4.2.1 64KB を Mid 扱いに変更した追加検証 (OOM 対策)

64KB を Large (mmap) から Mid に移した結果、full-range 条件での OOM が解消され、T=8/R=90 で +10.3%, T=16/R=90 で +4.6%, T=16/R=0 で +4.2% の改善を確認した。詳細は docs/benchmarks/2026-01-23_hz3_64kb_mid_fix.md にまとめた。

4.4 ST dist_app (RUNS=3)

表 5 を参照。ST では tcmalloc が最速で、hz3 が僅差で続く。

4.5 RSS (ru_maxrss)

表 6 と表 7 を参照。hz3 は MT remote と ST dist_app の双方で最小 RSS を示した。

4.6 メモリ解放チューニング比較 (Appendix)

表 8 に、SSOT 条件 (T=16/R=90, RUNS=10) で各アロケータの節約設定を比較した結果を示す。mimalloc の purge=0 は RSS が最小 (0.52GB) かつ 112M ops/s を維持した。一方、hz3 の S64 ON は baseline よりスループット / RSS とともに僅かに悪化しており、S64 は「常に有利」ではなくトレードオフであることが分かる。S64 Light のチューニング (A: purge delay=8) はデフォルト S64 より RSS を改善しつつ 104M ops/s を維持した (詳細は docs/benchmarks/2026-01-24_S64_LIGHT_TUNING_RESULTS.md)。

4.7 Larson sanity (S62 OFF)

Larson hygienic では hz3 が 143.1M ops/s を記録し、tcmalloc と同等水準で動作することを確認した（詳細は結果ログ）。

4.8 mimalloc-bench subset (RUNS=5, Appendix)

表 9 を参照。xmalloc-testN / sh8benchN / cache-scratchN では hz3 が最速、alloc-testN では hz4 が最速となった。

4.9 Redis workload (RUNS=3, Appendix)

表 10 を参照。5 パターン中 3 つで hz3 が最速。

4.10 最新スナップショット (2026-02-18, 追補)

本稿執筆後に、4 allocator (hz3/hz4/mimalloc/tcmalloc) で MT lane x remote% 行列 (RUNS=10) と redis-like 実ベンチ (RUNS=10) を再取得した。結果は以下の通りで、**hz3 と hz4 の補完関係**がより明確になった。

- main_r0 (local-heavy) : hz3 が最速 (375.4M ops/s)
- main_r90 (remote-heavy) : hz4 が最速 (67.6M ops/s)
- cross128_r90 (高スレッド remote) : hz4 が大差で最速 (50.65M ops/s)
- redis-like (memtier 15s, RUNS=10) : hz3 571,199 / mimalloc 568,740 / tcmalloc 568,052 / hz4 560,576

運用上は、既定を **hz3**、remote-heavy / high-thread 条件では **hz4 プロファイル** を選択するのが現時点で最も安定な方針である。

なお、hz4 は以前 redis preload 時に segfault (rc=139) があったが、malloc_usable_size の interpose 修正後、redis-like RUNS=10 で n_ok=10 を確認した。

4.10.1 プロファイル選択の最小指針

再計測結果 (RUNS=10) から、実運用向けには次の選択が妥当である。

条件	推奨プロファイル	根拠 (2026-02-18)
local-heavy (main_r0)	hz3	375.4M (hz4 137.4M)
remote-heavy (main_r90)	hz4	67.6M (hz3 62.6M)
high-thread remote (cross128_r90)	hz4	50.65M (hz3 1.80M)

4.11 実験条件

- CPU: 16-core x86_64
- OS: Ubuntu (native)
- コンパイラ: GCC (system default)、最適化はビルド既定
- ビルド構成: hz3 scale lane / hz4 default
- ラン数: RUNS=10 (中央値採用)、旧付録データの一部は RUNS=3/5

- 比較対象: mimalloc、tcmalloc (gperftools)、system (glibc)

再現手順:

- docs/benchmarks/2026-01-24_PAPER_BENCH_WORK_ORDER.md の手順に従う。(SSOT / T sweep / R sweep / dist_app / RSS / Appendix を含む)

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

R=90 (T=16) では hz4 が最速であり、message-passing による remote-heavy 最適化が有効に機能する。一方、R=0 では hz3 が最速であり、local-only での固定費が小さいことが示された。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

- **ST dist_app**: tcmalloc が最速、hz3 が僅差で続く
- **Larson sanity**: hz3 は tcmalloc と同等水準
- **redis-like (追補)**: hz3/mimalloc/tcmalloc がほぼ同水準で、hz4 も実用レンジに入る

これらの結果から、hz3 は実ワークロードでも競合可能な水準にあることが確認できた。

一方、mimalloc-bench subset ではベンチにより勝ち負けが分かれ、速度とメモリ効率のトレードオフが明確に現れた。

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

hz4 は T=16 で大きく伸びる一方、ST や local-heavy では不利なケースが残る。これは欠陥というより、remote 専用に最適化した設計上のトレードオフである。現時点では用途別プロファイル運用 (hz3/hz4 分離) が合理的であり、将来的な統合最適化は別課題として扱う。

- 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

5.6 運用指針 (現時点)

- 既定: hz3 (local-heavy、redis-like、低 RSS 重視)
- 切替: hz4 (remote-heavy、高スレッド cross 条件)
- 研究開発: hz4 の再挑戦は小箱ではなく大箱 (構造変更) で進める

6. 限界と今後の課題

- 本稿は Ubuntu (native) の結果を主として採用する。
- PTAG32 routing は free の分岐を単純化する一方、tag 参照により **L1 ミスが増える傾向**がある。ヘッダ方式とのトレードオフは残るため、用途に応じてビルド切替を行う。
- 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 (Redis など) 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、local-only 条件で最速かつ RSS 最小を示した。さらに、後発の hz4 を remote 専用プロファイルとして分離したことで、single-profile では取りにくい high-thread/remote 条件の性能を実用域まで引き上げた。今後は、dual-profile 運用を維持しつつ、統合最適化の可能性を大箱で検証する。

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> <https://zenodo.org/records/18674502> <https://doi.org/10.5281/zenodo.18674502>

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 2: SSOT (T=16, R=90, RUNS=10, median; M ops/s).

Allocator	hz3	hz4	mimalloc	tcmalloc
ops/s	71.6	81.7	72.2	77.8

Table 3: T sweep (R=90, RUNS=3, median; M ops/s).

T	hz3	hz4	mimalloc	tcmalloc
4	83.1	101.6	75.6	78.4
8	80.8	73.2	75.9	66.7
16	76.6	106.3	85.0	79.5

Table 4: R sweep (T=16, RUNS=3, median; M ops/s).

R	hz3	hz4	mimalloc	tcmalloc
0%	359.6	249.6	300.5	357.0
50%	94.4	97.4	84.6	103.3
90%	75.3	97.3	75.1	74.9

Table 5: ST dist_app (RUNS=3, median; M ops/s).

Allocator	ops/s
tcmalloc	80.2
hz3	75.2
mimalloc	73.4
hz4	51.6

Table 6: RSS MT remote (T=16, R=90; GB, lower is better).

Allocator	Max RSS (GB)
hz3	1.36
mimalloc	1.52
hz4	2.04
tcmalloc	2.34

Table 7: RSS ST dist_app (KB, lower is better).

Allocator	Max RSS (KB)
hz3	3456
mimalloc	4480
tcmalloc	7936
hz4	14464

Table 8: Memory-release tuning (SSOT T=16, R=90, RUNS=10, median; M ops/s / GB).

Allocator (mode)	ops/s	RSS (GB)
hz3 (S64=0)	115.2	1.39
hz3 (S64 ON)	111.7	1.48
mimalloc (baseline)	102.1	1.12
mimalloc (purge=0)	112.1	0.52
tcmalloc (baseline)	63.8	2.79
tcmalloc (release_rate=10)	62.9	2.85
jemalloc (decay=0/0)	106.9	0.12

Table 9: mimalloc-bench subset (RUNS=5, median; ops/s).

Benchmark	hz3	hz4	mimalloc	tcmalloc
alloc-testN	24064	89856	13952	17664
xmalloc-testN	98048	71424	92232	44028
sh8benchN	239268	161088	159152	128588
cache-scratchN	9856	4600	3712	7680

Table 10: Redis workload (RUNS=3, median; M ops/s).

Pattern	hz3	hz4	mimalloc	tcmalloc
SET_ADD	2.59	2.22	2.46	2.31
GET	2.75	1.89	2.43	2.23
LPUSH	2.48	2.40	2.59	2.15
LPOP	2.42	2.15	2.59	2.07
RANDOM	2.32	1.93	2.20	2.22