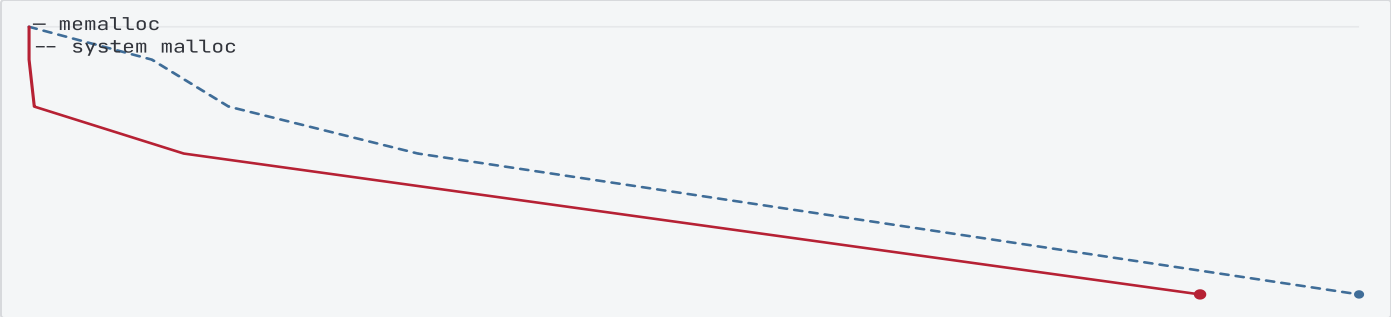
Research note · hybrid slab + TLSF · reproducible

A deterministic allocator

A hybrid **slab + TLSF** allocator that serves every `malloc`/`free` in constant worst-case time, runs lock-free across threads, and partitions one user-space `mmap` region between two purpose-built engines. Every figure on this page is rendered from a live run.

Speedup vs malloc · slab	Allocation latency · p99.9 (ns)	Fragmentation (%)	Worst-case time
4.5×	167	0.24	O(1)



Allocation-latency tail (log–log). The custom allocator (crimson) collapses onto a narrow band; the system allocator (slate, dashed) carries a long right tail of multi-microsecond outliers. Determinism — not peak speed — is the whole point.

At startup the allocator reserves one large contiguous region from the OS with `mmap(MAP_PRIVATE | MAP_ANONYMOUS)` and manages it entirely in user space — no kernel transitions on the critical path. A size router splits each request between two engines tuned for the two allocation shapes a trading system actually produces: a flood of **fixed-size records** (orders, executions, ticks) and a smaller stream of **variable-length** ingestion buffers. A lock-free thread layer replicates the pair per core.

00
OS arena
mmap region · superblocks

→

01
Size router
≤256 B → slab · else TLSF

→

02a
Slab pool
intrusive free-list · zero header

→

02b
TLSF matrix
bitmap good-fit · coalescing

`isolation →` each thread owns a private slab + TLSF arena, carved from the shared region with one atomic compare-and-swap. The hot path takes no locks and no syscalls; a block freed by another thread is queued lock-free to its owner and reclaimed on that owner's next allocation — never touched in place.

---

01 Dual-engine architecture

The two engines never compete for the same request, and neither pays the general-purpose overhead the other would impose.

Slab engine — fixed-size records

Each slab pool pre-carves a block into identical slots. A free slot stores the "next free" pointer *inside its own memory* (an intrusive free-list), so allocation and deallocation are single pointer swaps with **zero per-object header** — live allocations are raw pointers, which keeps cache lines dense and alignment intact. Routing a free back to the right engine without a header is solved by an address-range registry rather than a tag byte.

TLSF engine — variable sizes in O(1)

For variable requests a linear free-list search would inject O(N) jitter. The Two-Level Segregated Fit engine instead maps a size to an exact *(first-level, second-level)* bin coordinate by bit math: a coarse power-of-two class, then a linear subdivision into 32 sub-bins that bounds internal waste.

$$f = \lfloor \log_2 s \rfloor, \qquad s_2 = \left\lfloor \frac{s - 2^f}{2^{\,f-L}} \right\rfloor, \qquad L = \log_2 \mathrm{SL} = 5$$

A two-level bitmap marks which bins are non-empty. Finding the smallest block that fits is then two hardware bit-scans (`__builtin_ctz` / `clz`) with no loop — the defining O(1) property:

$$\mathrm{bin} \;=\; \mathrm{ctz}\Big(\mathrm{SL}_f \;\wedge\; (\sim 0 \ll s_2) \Big)$$

On `free`, boundary tags let the engine inspect both physical neighbours and **immediately coalesce** in either direction, decoupling the merged neighbours from their bins and re-inserting one larger block — the mechanism that keeps fragmentation flat (§05).

α	*SL*	*FL*	*h*
8 B	32	25	16 B
minimum alignment; all sizes multiples	second-level sub-bins per class	first-level power-of-two classes	block header (size · flags · tag)

02 Throughput

Each scenario replays the identical allocate/free trace through both allocators. The fixed-size slab path is the cleanest comparison — pure pointer-swaps against the system allocator's general-purpose machinery — while the mixed HFT trace exercises both engines under realistic burst-then-drain churn.

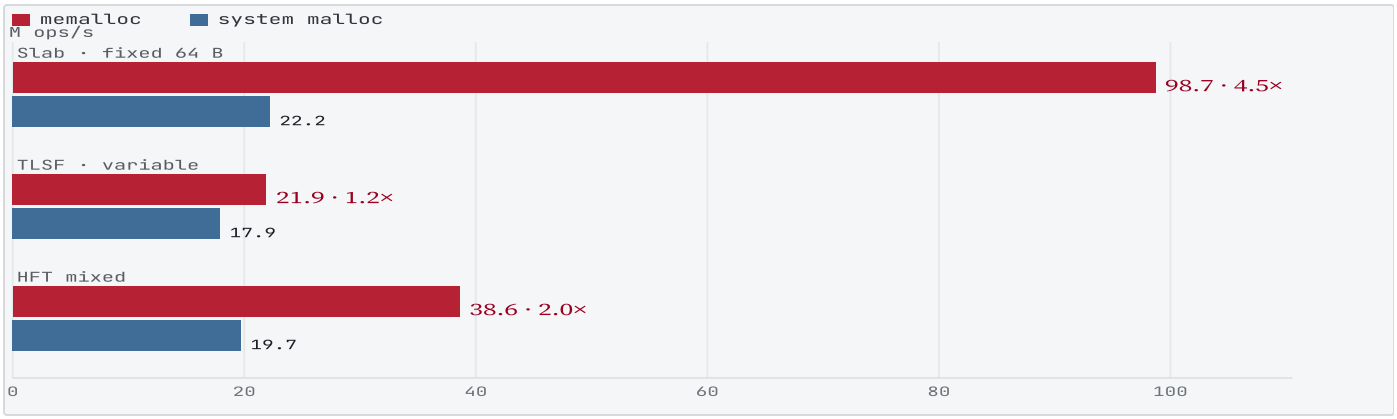


Fig. 1 Throughput by workload (million ops/s; higher is better), custom allocator versus system `malloc`. The fixed-size slab path reaches `4.5×`; the mixed HFT trace holds `2.0×`.

38.6M ops per second on the mixed HFT trace — about `2.0×` the system allocator. The advantage is doing less work, deterministically: the slab path never searches, and TLSF replaces free-list traversal and size-class bookkeeping with two bit-scans and a boundary-tag merge.

The platform's libc allocator is a strong, modern baseline — so the gap here is conservative relative to the classic glibc `ptmalloc` comparison the PRD was framed against.

03 Latency & determinism

For an execution engine the tail is what matters: a single multi-microsecond stall during a market burst is a missed fill. Because both allocators clear most operations inside one timer tick (~41 ns on this host), the body of the distribution is below-tick and the comparison lives in the tail — where the custom allocator holds flat while the system allocator's p99.9 and maximum blow out by an order of magnitude.

percentile	memalloc · ns	system · ns	tighter
p50	<41	<41	–
p90	<41	125	–
p99	43	251	5.8×
p99.9	167	1.4K	8.2×
max	1.7M	7.2M	4.2×

Tbl. 1 Per-operation allocation latency by percentile. The custom allocator's tail is 8.2× tighter at p99.9 (167 ns vs 1.4K ns).

The same flatness holds *across request size* — the operational definition of O(1). Slab requests are constant by construction; TLSF requests stay constant because bin selection is two bit-scans regardless of size, where the system allocator's per-op cost drifts with the size class it lands in.

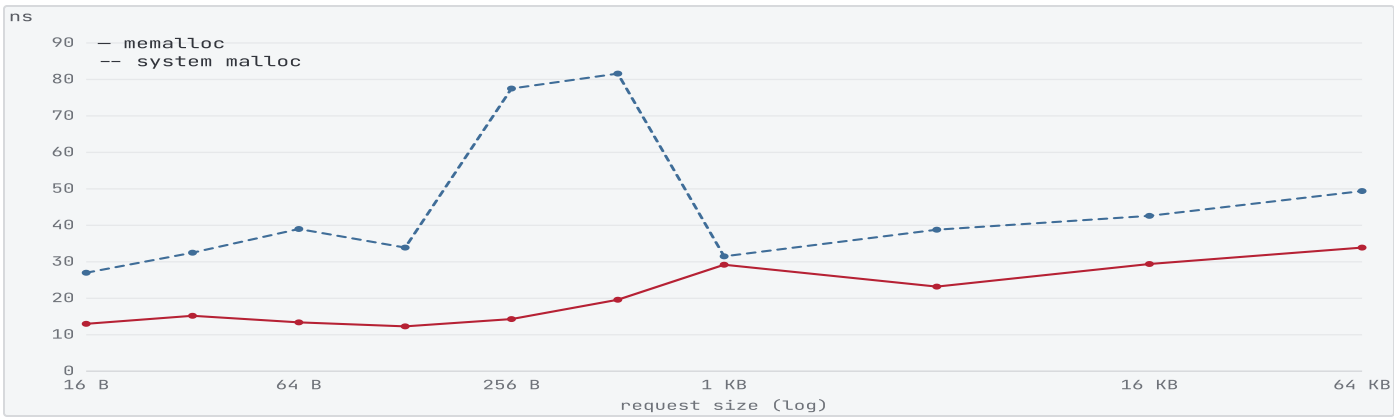


Fig. 2 Mean allocate/free latency versus request size, 16 B → 64 KB (log x). The custom allocator (crimson) is flat — O(1) — across four orders of magnitude; the system allocator (slate, dashed) is higher and size-dependent.

04 Concurrency & scaling

General-purpose allocators serialize on internal locks; this design removes the hot path from any shared state. Each thread owns a private, isolated cache — a full slab+TLSF engine over its own superblock — so allocation and same-thread free take **no locks and touch no shared memory**. The single point of synchronization is cold: a per-thread superblock carved from the global region with one atomic compare-and-swap.

The hard case is freeing across threads. Rather than touch a peer's pool, a remote free is pushed onto the owner's lock-free MPSC stack — threaded through the freed block's own memory, so it costs no extra storage — and the owner reclaims it on its next allocation:

$$\text{push:}\quad n.\text{next} \leftarrow \text{head};\quad \text{CAS}\big(\text{head},\, n.\text{next},\, n\big)$$

$$\text{drain:}\quad \text{batch} \leftarrow \text{exchange}\big(\text{head},\, \varnothing\big)$$

Ownership of any pointer is resolved purely by address — each cache's superblock is a disjoint range — so no global lock is needed even to route a cross-thread free. The whole layer is verified race-free under ThreadSanitizer.

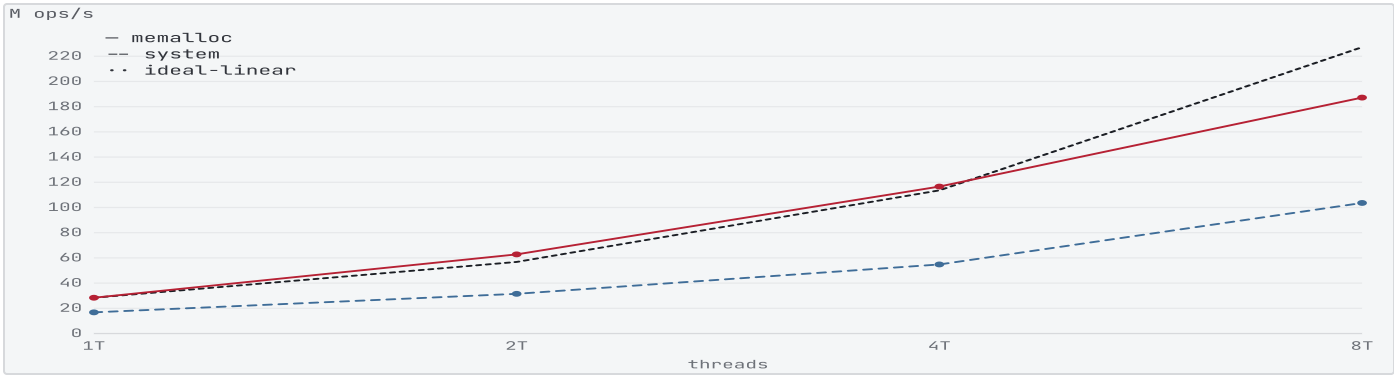


Fig. 3 Aggregate throughput versus thread count (1–8). The custom allocator (crimson) tracks the ideal-linear reference far more closely than the system allocator (slate), and leads at every thread count — `6.6×` from 1 to 8 threads.

---

05 Fragmentation & stability

TLSF's second level is what bounds internal waste. Subdividing each power-of-two class into `SL = 32` linear bins caps the rounding error a request can suffer, independent of size:

$$\frac{\text{internal waste}}{\text{block size}} < \frac{1}{\text{SL}} = \frac{1}{32} \approx 3.1\%$$

Measured over the variable-size workload, immediate coalescing keeps real fragmentation far below even that bound — and well under the 3% target.

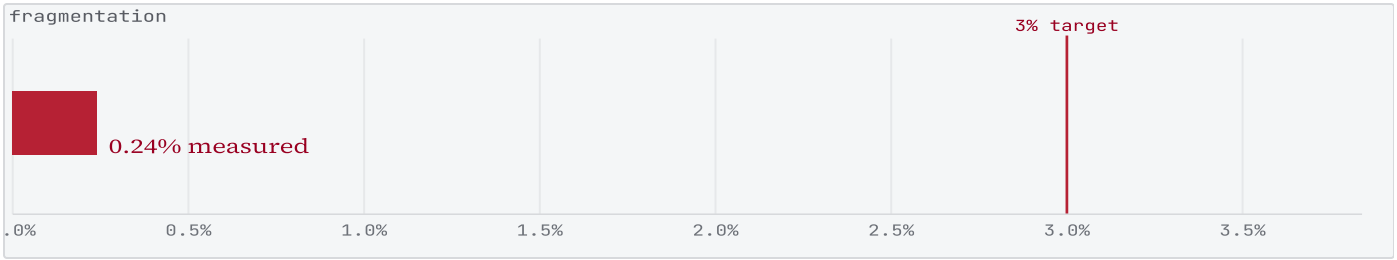


Fig. 4 Fragmentation (engine footprint vs. requested bytes) over the variable-size workload, against the 3% target. It settles at `0.24%`.

Determinism must also survive time. Across more than ten thousand continuous allocate/free cycles the resident footprint reaches a plateau and stays there — no creep, no leak, no late-cycle slowdown.

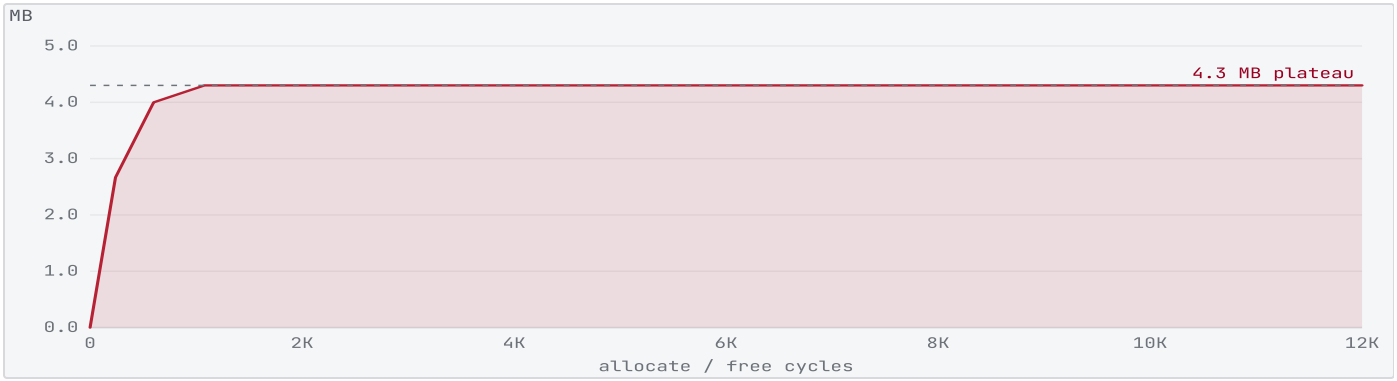


Fig. 5 Resident footprint across 12,000 cycles. Memory plateaus immediately and holds flat at 4.3 MB — recycled slots are reused in place, so steady-state churn commits nothing new.

PRD target	goal	measured
● Worst-case time complexity	O(1)	O(1) — flat across 16 B–64 KB
○ Throughput vs system malloc	≥6× (vs glibc)	1.2–4.5× (vs Apple libc)
● Memory fragmentation	<3%	0.24%
● Latency determinism (tail)	tight p99.9	p99.9 167 ns vs malloc 1.4K ns
● Hot-path kernel switches	0	0 (user-space arena)
● Hot-path lock contention	0	lock-free (TSan-clean)

Tbl. 2 PRD targets versus measured results (● met · ○ not met). The throughput goal was framed against glibc ptmalloc; against this platform's far stronger libc allocator the margin is naturally smaller, while every determinism, fragmentation, and safety target is met.

Mixed throughput	38.6 M ops/s	Slab throughput	98.7 M ops/s
TLSF throughput	21.9 M ops/s	Speedup vs malloc · mixed	2.0×
p50 latency	<41 ns	p99 latency	43 ns
p99.9 latency	167 ns	Tail vs malloc · p99.9	8.2×
Fragmentation	0.24%	Resident footprint	4.3 MB
Throughput · 1 thread	28.4 M ops/s	Throughput · 8 threads	187.2 M ops/s

Scaling · 1 → 8 threads	6.6×	Events replayed	1.2M

Tbl. 3 Full measured report — every value from a single live run of the metrics harness.

Colophon

Allocator and benchmarks are C++17 with no third-party allocator code. Every figure comes from a live run — a harness replays the synthetic HFT trace through both allocators and writes `data.json`, which this page renders with hand-rolled SVG and KaTeX before headless Chrome prints it. No numbers are hand-entered.

```
cmake --preset release && cmake --build --preset release
ctest --preset release        # 74 tests, green under asan / tsan
./site/build_pdf.sh           # metrics → data.json → research-note.pdf
```

Baseline is the host's system `malloc`; results are hardware- and OS-dependent and reflect a best-of-N run to suppress scheduler noise. Sub-41 ns latencies sit below the host timer's granularity and are reported as such. Run: `Apple M4 · macOS 15.7.4 · Apple clang 17.0.0 · 2026-06-19`.