LOCUS: Owner-Drained Chunk Mailboxes for KV-Block Recycling in CPU LLM Inference

Victor Bona
Independent Researcher
Software Engineer
Blumenau, Brazil
victor.bona@hotmail.com

***Abstract*—LOCUS is a domain memory pool for CPU large-language-model (LLM) serving whose remote-free path is a per-worker lock-free chunk mailbox: a finished request's key-value (KV) blocks are returned as one atomic push, and the pool owner drains every mailbox off the allocation hot path, so freeing never contends on a shared queue and the design carries zero tuning parameters. On LOCUS-EVAL v1, a frozen four-workload suite of deterministic serving-shaped KV traces run against jemalloc, mimalloc, and system malloc on Apple Silicon, the mailbox ranks first on velocity on every workload. A post-publication correctness audit found that the frozen suite let the pool skip a per-block memory write the malloc baselines paid, so the audited, touch-parity margins are the headline: at a one-byte-per-block touch parity LOCUS is 1.6x faster than mimalloc on steady decode, 2.7x on burst cancellation, and 2.3x on long-tail decode, and the margin compresses to about 1.15x over system malloc when full KV writes make memory bandwidth dominate. The transient cost that ships with the win is a burst-cancellation footprint of up to 1.5x the theoretical peak block count, because the owner drains once per step. The scope is deliberately narrow: a single host, macOS on Apple Silicon, synthetic deterministic traces rather than a live serving engine, and an allocator-level measurement. NUMA locality is designed and syscall-validated but its benefit is unmeasured, and end-to-end serving integration is future work. The evidence supports one claim: as a fixed-size KV-block pool under realistic concurrent churn, the parameterless owner-drained mailbox is faster than three general-purpose allocators at touch parity while deleting an entire tuning surface.**

***Index Terms*—KV cache, memory pool, remote free, lock-free data structures, LLM inference, CPU serving, NUMA locality, allocator benchmarking, Rust, reproducible evaluation.**

I. INTRODUCTION

A CPU LLM serving engine spends a large and recurring share of its memory traffic on the KV cache: fixed-size blocks that hold the per-token attention state of in-flight requests. Requests arrive, prefill a burst of blocks, decode one block at a time, and then finish or cancel, at which point their blocks must be returned for reuse. This is not a general allocation workload. The block size is fixed, the live set is bounded by a serving budget, and the lifetime pattern is dominated by request-shaped groups of blocks that are born together and freed together. The question this paper answers is narrow and measurable: given that shape, what is the fastest way to recycle KV blocks across the worker threads that produce and consume them on a single host, and how much faster is it than simply calling a general-purpose allocator?

LOCUS answers with a domain pool rather than a global allocator replacement. That choice is recorded as an architecture decision (ADR 0001): the serving workload has several distinct memory classes (request metadata, private and shared-prefix KV cache, mapped weights, scratch tensors, pinned staging), and starting from an explicit domain pool keeps attribution, placement policy, and benchmark isolation tractable in a way that replacing the process allocator would not [5]. Inside that pool, the recurring cost is the cross-thread free: a worker that finishes a request must hand its blocks back to whatever owns the free list, and the obvious mechanism, a shared bounded queue, turns out to be the wrong one under real concurrency.

The contribution is threefold. First, a design: a parameterless per-worker chunk mailbox for owner-drained remote frees, with a flat last-in-first-out (LIFO) pool free list and generation-validated block handles, arrived at by removing tuning surface rather than adding it. Second, a method: a falsification-first evaluation discipline in which every change is a numbered, written prediction measured before the code is trusted, results are taken from at least two runs (a third when the first two disagree by more than five percent), and falsified predictions are kept in the record unedited. Third, an artifact: LOCUS-EVAL v1, a frozen, committed allocator benchmark against jemalloc, mimalloc, and system malloc, published with a self-critical touch-parity audit that corrected its own headline downward. The intended reader is an engineer or researcher evaluating KV-cache memory management for CPU inference, or building on the committed crate [1].

II. DESIGN GOALS AND CONSTRAINTS

LOCUS is shaped by a small number of constraints that are recorded as architecture decisions rather than left implicit. Table I lists them with their engineering effect. Two are worth stating plainly. The project is safe by default: the workspace forbids unsafe code, and exactly one module, the `sys` module, holds a scoped allowance for the `mmap` and `mbind` calls and raw memory handles that owned mappings require, with SAFETY comments on every unsafe block (ADR 0002) [6]. And the free path is off the allocation hot path: workers never touch the pool's free list directly; they publish to a mailbox and the owner reclaims, so allocation and freeing do not serialize against one another.

TABLE I
PRIMARY DESIGN GOALS AND CONSTRAINTS

Constraint	Engineering effect
Domain pool, not a global allocator (ADR 0001)	KV memory class, placement, and lifetime stay visible; benchmarks isolate the pool from process-wide allocation.
Safe by default (ADR 0002)	The crate denies unsafe code; only the `sys` module allows it, for owned mappings and Linux memory policy, under focused review.
Parameterless remote-free path	The mailbox has no capacity, batch limit, or retune machinery; there is nothing to tune per deployment.
Owner-drained frees	Workers push whole request chunks; the owner drains off the allocation hot path, so freeing never contends on a shared queue.
Reproducible frozen evaluation (ADR 0003)	Standing suites are frozen on publication; validation gates live in a separate crate so lower layers stay narrow.

III. ARCHITECTURE

LOCUS is published as a single Rust crate, `locus-alloc` (import name `locus_alloc`), version 0.1.0, with the KV pool and the mailbox as its two public performance-bearing types [2]. The remaining allocator experiments (scratch arenas, mapped and pinned scratch, placement probes) survive as internal, documentation-hidden modules.

A. The KV block pool

`KvBlockPool` is a fixed-size block pool tagged with an intended NUMA node. Every block is the same size (4 KiB in the evaluation), and the pool owns its storage through one of two backings: one heap allocation per block, or one contiguous mapped region in which blocks are fixed offsets. A block is referenced by an opaque `KvBlockHandle` carrying an index and a generation counter. Allocation pops a free index; freeing increments that index's generation, so a stale handle to a since-reused block is rejected on validation rather than silently aliasing live memory. The free list is a flat double-ended queue reused in LIFO order by default: the most recently freed block is handed out first, which keeps its cache lines warm (Section V). The pool tracks allocation and free counts and a high-water mark, which the evaluation uses to measure peak outstanding blocks exactly. A companion `KvBlockTable` maps one sequence's logical token positions onto its physical block handles, appending blocks as the sequence grows and releasing them all back to the pool on completion; this is the block-table indirection that PagedAttention popularized [8], kept explicit so that placement and lifetime remain visible per sequence.

The pool can also be backed by one contiguous mapped region rather than per-block heap allocations, which lets the entire pool be bound to a single NUMA node with one `mbind` call (`bind_to_node`) and its address span reported for placement verification (`mapping_span`). The mapped backing costs nothing on the write-touch hot path relative to heap blocks (Section V), and the bind path is the syscall-validated but as-yet-unmeasured foundation of the NUMA locality work (Section IX).

B. The chunk mailbox

`ChunkMailbox<T>` is the remote-free transport, and it is deliberately the smallest structure that can do the job: a single atomic pointer that is the head of a Treiber stack. A producer wraps a chunk in a node and publishes it with one compare-and-swap; the owner takes the entire pending list with one atomic swap, reverses it so delivery is oldest-first per producer, and reclaims each node exactly once. There is no capacity bound, no occupancy accounting, no blocking protocol, and therefore nothing to size or tune. Cloneable sender handles let every worker hold a producer for its own mailbox. The type carries the crate's only performance-relevant unsafe code, three blocks with SAFETY comments in `sys/chunk_mailbox.rs`, justified by exclusive list ownership after the swap and single-reclaim of each node [1].

C. The owner-drain loop and the free ABI

The two types compose into a serving-engine embedding pattern, shipped as an example [7]. One owner thread owns the pool. Each worker thread owns one mailbox sender. When a request finishes or cancels, the worker frees the request's whole block vector with one mailbox push; it performs no per-block frees and touches no shared queue. Once per scheduling step the owner drains every mailbox with `take_all` and returns the drained handles to the pool free list. The free ABI is therefore request-chunk-granular end to end: blocks travel

from worker to owner in the request-sized groups they were born in, one queue operation per request rather than one per block. This is the interface the evaluation exercises and the one a serving engine is expected to adopt.

IV. Falsification-First Methodology

The evaluation method is itself a contribution, because it is what makes the numbers in this paper trustworthy. Every change to LOCUS begins as a numbered postulate: a falsifiable prediction, written before any code or measurement. The matching experiment records the harness, at least two independent benchmark runs, and a verdict stated against the prediction. When the first two runs disagree by more than five percent the whole benchmark binary is rerun a third time, because concurrent microbenchmarks have real cross-run variance and a single run must not be trusted for ranking (a lesson learned the hard way in experiment 0351). Falsified postulates are kept and labeled, never rewritten; a prediction being wrong is data. The research record is immutable: no file in the experiment, postulate, evaluation, or dev-note history is edited after the fact, including filenames, and the raw Criterion logs that survived their sessions are committed verbatim (thirty logs under `documentation/evaluations/logs`) [4].

The five-percent rule is not decorative; it fires. In LOCUS-EVAL v1 it triggered a third run for *locus-mailbox* on burst-storm, long-tail, and churn-touch, and for system malloc on burst-storm and churn-touch, the whole benchmark binary rerun each time. Those are exactly the concurrent workloads where a single run's confidence interval was widest, and the third runs tightened them; a scoreboard built from single runs could have reported a different order on at least one workload. Proof gates are first-class alongside timing: a run whose accounting does not balance (allocated not equal to freed, mailbox submitted not equal to drained, a shared-queue counter off, or any disconnect) is not a slow result but an invalid one, and is discarded rather than ranked.

The discipline earns its keep by rejecting things. A large capacity-and-retune apparatus for the bounded remote-free queue, built across many experiments with dry-run planners, guarded mutation limits, and telemetry rollups, was obsoleted wholesale when the mailbox design removed the capacity it was tuning; the code was deleted with recovery hashes preserved in a code graveyard note. An even longer telemetry-rollup thread produced layer upon layer of JSON validation machinery and never yielded an allocator improvement; its intentionally unwieldy repeated-suffix filenames are preserved as an honest record of a runaway abstraction spiral. And a specific performance idea, chunk-preserving pool recycling, was falsified outright by experiment 0356 and its APIs reverted. A method that only ever confirmed its predictions would prove nothing; this one visibly kills its own work.

V. The Design Thread

The shipped design is the end of a short evidence chain, experiments 0351 through 0360, each answering the previous one's open question [4]. The thread is worth walking because the obvious design fails first, and the reasons it fails are what the final design is built to avoid.

0351: the obvious shared queue collapses under real concurrency. Earlier single-threaded sweeps had suggested that a large drain batch was the important lever for a shared bounded remote-free queue. The first benchmark in which enqueue and drain genuinely overlap across threads falsified that: batch size became statistically irrelevant, and producer count dominated instead, with cycle time rising roughly 5.3 μs at one producer to about 21 μs at four as send-side contention on the shared channel grew. Single-thread microbenchmarks had hidden the contention entirely.

0352 and 0353: sharding and chunking, but a tuning surface remains. Giving each producer its own queue removed the contention term and bought about 1.63x at four producers, but left a coordination floor that sharding could not touch. Publishing each producer's frees as one whole chunk instead of per-handle entries removed a per-item cost worth about 2 μs per cycle. Together the two levers took four-producer cycle time from about 21 μs to 10.8 μs, but the design still carried a bounded-queue capacity to choose.

0354 and 0355: a realistic trace, and pooling with a bad handoff loses to every malloc. A deterministic mixed-lifetime KV trace, the workload shape a serving engine actually produces, confirmed the sharded chunk path at 80 μs per trace against 282 μs for the shared per-handle path. Run against jemalloc, mimalloc, and system malloc freeing natively on worker threads, the pooled sharded-chunk design led mimalloc by about 2.5x on this untouched trace. The sharpest lesson was in the other direction: the shared per-handle pool path (282 μs) was slightly slower than plain jemalloc. A KV pool with a naive shared free queue is worse than not pooling at all; the win comes specifically from the chunk-shaped handoff, not from pooling.

0356: chunk-preserving recycling, falsified and kept. The natural next idea, preserving request chunk identity inside the pool with dedicated `free_chunk`/`allocate_chunk` paths, predicted a drop below 60 μs. It gained nothing (the chunk-pool path was indistinguishable from the flat path) and, worse, merely carrying the extra recycled-chunk store regressed the untouched hot path about eleven percent. The locality argument was void on reflection: a flat LIFO free list already returns drained chunks warm and burst-shaped. The change was reverted; the pool keeps its single flat LIFO free list. Chunk identity pays at the transport layer and stops paying at the pool boundary.

0357: the parameterless mailbox, the shipped design. With pool bookkeeping ruled out as the bottleneck, the remaining cost was the bounded-queue protocol itself. Replacing per-worker bounded chunk queues with minimal lock-free chunk mailboxes (Treiber push, swap-take drain, no capacity or stats protocol) beat the bounded chunk queue by about six to eight percent with visibly tighter confidence intervals: the bounded queue's run had an eleven-percent-wide interval while the mailbox stayed under one percent. Removing the capacity protocol removed a variance source, not just mean

cost. The qualitative result mattered more than the percentage: the mailbox has zero tuning parameters, deleting the entire capacity-retune surface that a long line of prior experiments had built and tuned, while beating the tuned configuration.

0358 and 0359: warmth and placement, measured not assumed. Once blocks are actually written, LIFO reuse beats oldest-first (FIFO) reuse by eleven to eighteen percent, confirming that recency-respecting reuse is a real write-bandwidth saving and constraining any future placement policy to treat recency as a hard constraint. Backing the pool with one contiguous mapped region costs nothing on the churn hot path (identical untouched, five to nine percent faster on touch churn once first-fault effects settle), and the whole pool binds to a NUMA node in one `mbind` call on Linux, a path exercised and error-classified in Section IX.

VI. Evaluation: LOCUS-EVAL v1

LOCUS-EVAL v1 is the standing scoreboard the design's claims rest on [3]. It is frozen: any change to its workloads, metrics, or contenders requires a v2, and the published document is never edited to reflect a changed suite. It can gain an addendum, and it has one (Section VII), but never an edited table.

A. Suite definition

Four deterministic workloads, defined in Table II, exercise five contenders through the identical trace schedule (a shared, byte-identical workload module): LOCUS's per-worker chunk mailbox on a LIFO pool (*locus-mailbox*), the earlier shared bounded per-handle queue on the same pool (*locus-shared*, capacity 1024, batch 64), and jemalloc, mimalloc, and system malloc freeing natively on worker threads. In every case four workers free on real threads, with no synthetic sleeps anywhere. Velocity is the Criterion median per trace over two runs, a third run when the first two disagree by more than five percent; stability is the confidence- interval width; quality is peak outstanding blocks over the theoretical instant-free peak; and proof gates (allocated equals freed, mailbox submitted equals drained, shared-queue counters balanced, zero disconnects) are hard gates. All benchmarks ran on Apple Silicon under macOS with Criterion flags `-sample-size 20 -warm-up-time 1 -measurement-time 3`.

B. Velocity

Table III reproduces the frozen v1 velocity record verbatim, and Fig. 1 charts a representative median per contender. *locus-mailbox* wins all four workloads on velocity. *locus-shared* is last or second-to-last on all four, the clearest confirmation of the 0355 lesson: pooling without the chunk-shaped handoff is worse than any general-purpose malloc.

C. Quality

Quality is the transient footprint: peak outstanding blocks divided by the theoretical peak an instant-free allocator would hold. Table IV and Fig. 2 report the worst observed ratio across runs. On long-tail and churn-touch every contender sits at or below 1.01. On steady-decode and burst-storm the mailbox and the shared queue overshoot, because the owner drains once per step while a malloc worker frees instantly: on burst-storm *locus-mailbox* transiently holds 1.50x the theoretical peak (and *locus-shared* 1.62x) while every malloc stays at or under 1.14x. A serving engine sized to the theoretical KV peak would need roughly fifty percent headroom to absorb a full burst cancellation under this drain cadence, or the pool's existing drain-on-allocation-failure fallback, which bounds the overshoot at the pool size. This quality cost ships with the velocity win and is stated as part of the design, not hidden.

TABLE II
LOCUS-EVAL v1 Workloads (deterministic; 4 KiB blocks)

Workload	Shape	Blocks	Peak
steady-decode	64 requests, 4 arrivals/step, 16-block prefill, one block/decode step, decode 16/32/48, every 4th cancels at 8	2688	1344
burst-storm	same requests, all 64 arrive at step 0	2688	1537
long-tail	4 arrivals/step, every 8th decodes 512 steps, other 56 decode 16	6016	4168
churn-touch	256 live 16-block chunks, 64 steps/cycle, each step frees oldest and allocates plus fully writes a new chunk	—	4096

D. Stability and proof gates

Proof gates passed for every contender on every workload: allocated equals freed each trace, mailbox submitted equals drained, shared-queue counters balanced, zero disconnects, and no workload was unsupported by any contender. On stability, *locus-mailbox* intervals were under two percent in the majority of runs, with two wide exceptions (one long-tail interval near eighteen percent, one churn-touch interval near thirteen percent) that the third runs tightened to under two percent; even the worst mailbox interval beats the best baseline median by 3.8x. mimalloc and jemalloc stayed under four percent everywhere, system malloc was the least stable baseline (an eleven-percent churn-touch spread), and *locus-shared* sat in the two-to-seven percent band.

E. Predicted versus observed

Table V is the falsification ledger for the suite. The rankings were predicted before the runs, and two predictions were wrong in instructive ways. Long-tail, named beforehand as the workload most likely to embarrass the design, instead produced its largest untouched margin, because long requests multiply allocations (6016 versus 2688 blocks) and the pool's pop-an-index path scales with allocation count far better than malloc thread caches holding a 16.5 MiB live set: the prediction was wrong in direction, in the design's favor. Churn-touch went the other way, and is discussed as its own result in Section VII.

TABLE III
LOCUS-EVAL v1 VELOCITY: CRITERION MEDIAN PER TRACE (μS), PER-RUN MEDIANS (FROZEN v1 RECORD)

Workload	locus-mailbox	locus-shared	jemalloc	mimalloc	system
steady-decode	86.9 / 84.9 / 81.4	290.9 / 280.7 / 274.0	233.7 / 227.6	201.2 / 197.3	229.9 / 228.7 / 232.6
burst-storm	37.5 / 39.5 / 37.6	307.9 / 293.9 / 294.3	203.6 / 210.2	147.2 / 145.2	176.6 / 166.9 / 173.7
long-tail	66.4 / 61.5 / 60.4	646.1 / 622.5 / 635.2	414.3 / 418.5	283.0 / 278.6	464.4 / 466.4 / 443.1
churn-touch	170.0 / 183.8 / 168.6	238.0 / 236.9 / 230.8	223.0 / 229.0	201.6 / 209.0	181.2 / 202.1 / 196.1

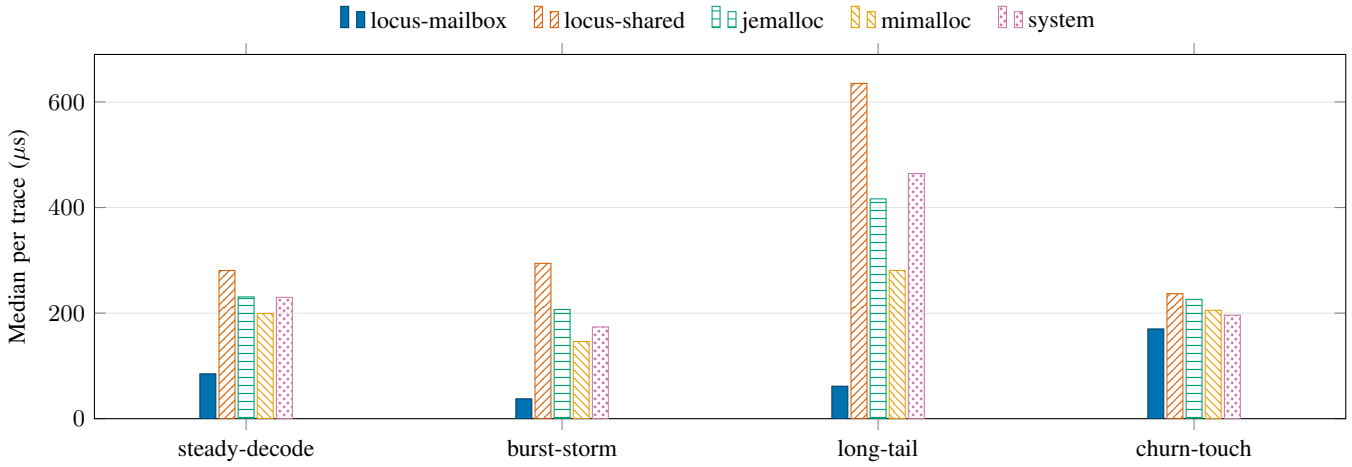

Fig. 1. LOCUS-EVAL v1 velocity, lower is better. Each bar is the median of the per-run medians listed in Table III (median of two or three runs). Fills combine a colorblind-safe hue with a distinct pattern so the series separate in grayscale. These are untouched-trace numbers for the three trace workloads; the audited touch-parity margins are in Fig. 3.

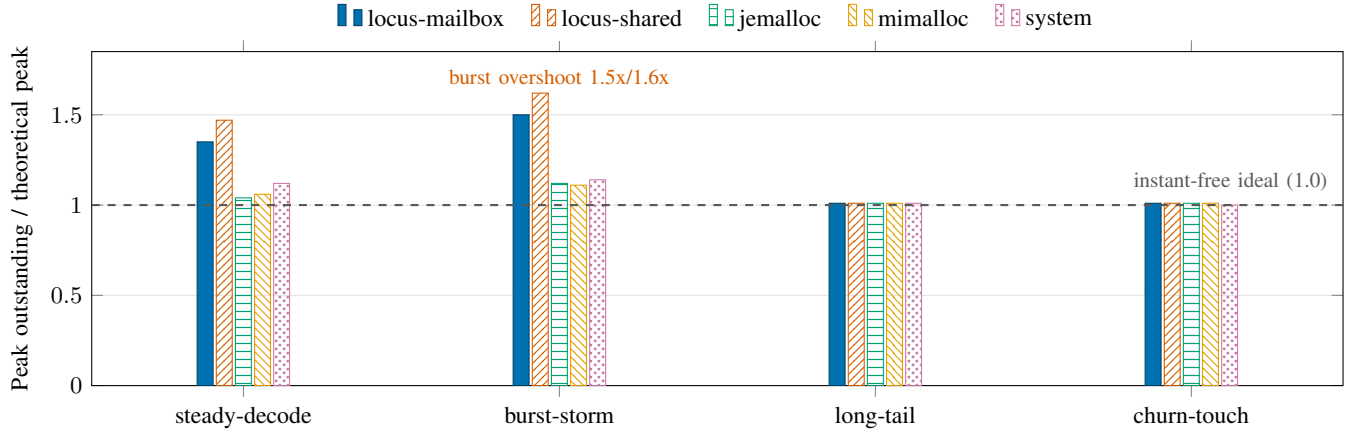

Fig. 2. LOCUS-EVAL v1 quality, closer to the dashed 1.0 line is better. The mailbox and shared paths overshoot only on the two arrival-clustered workloads, because the owner drains once per step; on long-tail and churn every contender is within one percent of ideal. Fills combine hue and pattern for grayscale.

F. Reproducibility

The suite is a committed artifact, not a description of one. The four contender binaries (`benches/locus_eval_{locus,jemalloc,mimalloc,system}.rs`) share a single versioned workload module (`benches/locus_eval/workloads.rs`) so that every contender runs a byte-identical trace schedule, and a common runner drives the malloc baselines through their native cross-thread free paths. Both the pool runner and the malloc runner time full free completion, and the pool backing is real pre-zeroed 4 KiB memory with generation-validated handles. The Criterion invocation is fixed at `-sample-size 20 -warm-up-time 1 -measurement-time 3`, and the raw Criterion logs that survived their sessions are committed verbatim so the published medians and quality ratios can be checked against them. The audit in Section VII confirmed that the published medians and quality ratios match the raw run logs exactly, that proof counters balanced in all runs, and that

TABLE IV
LOCUS-EVAL V1 QUALITY: WORST PEAK OUTSTANDING / THEORETICAL PEAK

Workload	mailbox	shared	jemalloc	mimalloc	system
steady-decode	1.35	1.47	1.04	1.06	1.12
burst-storm	1.50	1.62	1.12	1.11	1.14
long-tail	1.01	1.01	1.01	1.01	1.01
churn-touch	1.01	1.01	1.01	1.01	1.00

the trace schedules are byte-identical between runners.

VII. THE TOUCH-PARITY AUDIT

The most important number in this paper is the one the evaluation corrected about itself. A post-publication correctness audit of the harness found a real asymmetry in the three trace workloads: the malloc baselines write one byte into every allocated block (through `Vec::push`), while LOCUS's trace path never touched block memory at all. The malloc baselines were paying for cache warmth that LOCUS was silently skipping, so the untouched-trace margins overstated the real gap. This is reported here as a virtue of the method, not a footnote: the frozen v1 tables are left unedited, and a one-byte-per-block probe (run once, then reverted, never folded into the frozen suite) re-measured the trace workloads at touch parity.

Table VI and Fig. 3 give the correction. The rankings survive unchanged: *locus-mailbox* is still first on every workload. But the headline trace margins move. Over mimalloc at touch parity, the margins are 1.6x on steady-decode (from an overstated 2.3x untouched), 2.7x on burst-storm (from 3.9x), and 2.3x on long-tail (from 4.6x), the last being where half the untouched margin was cache warmth the baselines paid and LOCUS did not. The churn-touch control, where every block was already fully written, moved less than five percent, which validates the probe. This paper follows one rule as a result: the audited touch-parity margins are the headline everywhere, and the untouched margins (2.3x, 3.9x, 4.6x) appear only with the audit caveat in the same sentence, as here.

Two caveats cut in LOCUS's favor and are recorded with the correction. The probe pays per-write handle validation that the malloc `push` does not, so the touched numbers are an upper bound on the true cost; and a serving engine writes far more than one byte per block, a regime the churn-touch workload already prices, where the mailbox win over system malloc compresses to about 1.15x (169 versus 196 μs). The honest reading is that once real KV writes are counted, allocator choice is a single-digit percentage of end-to-end step time, and the large untouched multipliers are an artifact of an untouched microbenchmark.

VIII. RELATED SYSTEMS

LOCUS is measured against three general-purpose allocators and positioned as a domain pool, not a replacement for any of them. jemalloc [9] and mimalloc [10] are production allocators with per-thread caches and sharded free lists; mimalloc's sharded free-list design is the closest in spirit to what LOCUS does, and it is consistently the strongest baseline here, yet its general-purpose machinery (size-class lookup, page metadata, deferred remote-free segments) costs a measured 2.5x or more on fixed-size KV churn at touch parity. System malloc is the macOS default; its comparatively strong churn-touch showing is one reason the churn-touch workload and its narrow 1.15x result matter (Section IX). LOCUS does not compete on generality: it allocates one block size, from a pre-sized pool, for one memory class. Within that scope it can preserve caller-defined chunk identity end to end, from producer to free list, which no general-purpose allocator does.

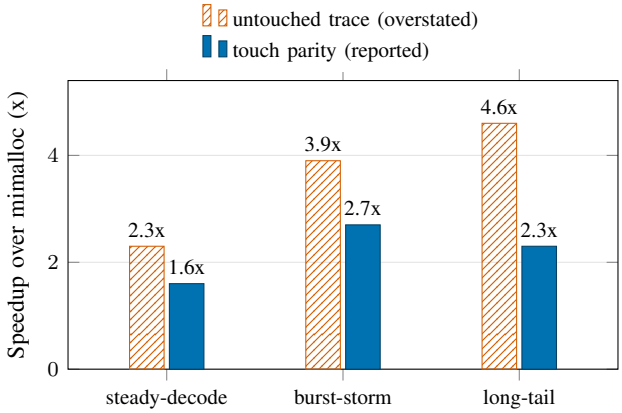

Fig. 3. The touch-parity correction, higher is a larger speedup. For each trace workload, the overstated untouched-trace margin over mimalloc (patterned) and the audited touch-parity margin actually reported (solid). The correction is largest on long-tail, where the untouched trace credited LOCUS with cache warmth the baselines were paying for.

The cross-thread free is precisely where both baselines spend their generality. jemalloc returns a block freed by a non-owning thread toward its owning arena and flushes it through per-thread caches on a decay schedule [9]; mimalloc keeps a per-page thread-free list that the owning thread reclaims lazily, its free-list sharding the same locality idea LOCUS applies [10]. Both therefore defer and batch remote frees, as LOCUS does. What neither can use is that a request's blocks were born together: to a general allocator they are N independent frees, each size-classed and threaded through page metadata on its own. LOCUS moves one pointer per request instead of touching N blocks, and that difference is the measured gap on fixed-size KV churn.

The motivating context is KV-cache memory management for LLM serving. vLLM's PagedAttention established block-structured, non-contiguous KV storage with near-zero KV fragmentation as the discipline that makes high-throughput serving practical [8], which is precisely why recycling those blocks cheaply across worker threads is worth optimizing. Fixed-size pooling itself is an old idea, slab and arena allocators being the classical instances [11]; LOCUS's contribution is not the pool but the parameterless owner-drained remote-free path layered on it, tuned to the request-chunk lifetime of KV blocks.

TABLE V
LOCUS-EVAL v1 PREDICTED (POSTULATE 0360) VERSUS OBSERVED RANKING. THE OBSERVED COLUMN IS THE FROZEN UNTOUCHED-TRACE v1 RESULT; THE LONG-TAIL 4.6X IS AN UNTOUCHED-TRACE MARGIN THAT THE TOUCH-PARITY AUDIT (SECTION VII) CORRECTS TO 2.3X OVER MIMALLOC.

Workload	Predicted	Observed
steady-decode	mailbox, mimalloc, jemalloc, system, shared	mailbox, mimalloc, jemalloc = system (tied), shared. Correct except a jemalloc/system tie.
burst-storm	mailbox, mimalloc, jemalloc, system, shared	mailbox, mimalloc, system, jemalloc, shared. System/jemalloc order inverted.
long-tail	same order, mailbox lead under 2x	mailbox, mimalloc, jemalloc, system, shared, untouched lead growing to about 4.6x (2.3x at touch parity, Sec. VII). Wrong in direction.
churn-touch	mailbox, mimalloc, jemalloc, system, shared; gaps 1.2x to 1.6x	mailbox, system, mimalloc, jemalloc, shared; mailbox over system only about 1.15x. Two misses.

TABLE VI
TOUCH-PARITY AUDIT (POST-PUBLICATION PROBE; v1 TABLES ABOVE REMAIN FROZEN AND UNEDITED)

Workload	locus-mailbox untouched (v1)	locus-mailbox 1-byte touch	Margin over mimalloc, touched
steady-decode	85 μs	124.9 μs	1.6x (was 2.3x)
burst-storm	38 μs	53.7 μs	2.7x (was 3.9x)
long-tail	61 μs	124.0 μs	2.3x (was 4.6x)
churn-touch (control)	169 μs	176.4 μs	unchanged within noise

IX. SCIENTIFIC LIMITS

The evidence is deliberately bounded, and the bounds are stated as prominently as the result. First, this is a single host: all LOCUS-EVAL v1 numbers come from one Apple Silicon machine under macOS. System malloc's strong churn-touch showing may be a macOS large-allocation artifact; a Linux rerun is required before the churn result is treated as general. Second, the workloads are synthetic and deterministic. They are the shape a serving engine produces (prefill bursts, steady decode, early cancellation), but they are traces, not a live serving engine under a real request distribution, and the allocation is measured in isolation from attention compute and scheduling. Third, and most consequential, the frozen v1 trace workloads were not touch-symmetric; the touch-parity audit (Section VII) corrects the headline, and touch-symmetric workloads are mandatory before any v2 result.

Fourth, NUMA locality is designed and syscall-validated but its benefit is unmeasured. The pool binds to a node in one `mbind` call, and the bind path was exercised end to end: on this host it reports `bind_unsupported_on_host`, and under an unconfined Linux container (OrbStack) the syscall returns `ENOSYS` because the VM kernel is built without `CONFIG_NUMA`. The path reaches the kernel and classifies its errors correctly, but a successful bind, and therefore any locality speedup, cannot be demonstrated on the available hardware. No cross-node locality number is claimed. Fifth, one latent harness bug is documented: the pool's `allocate_with_backpressure` discards drain counts, so if pool exhaustion ever triggered its internal drain the end-of-trace wait loop could hang. It provably did not affect v1: the measured peak outstanding (2304 of an 8192-block trace pool, 4144 of a 16384-block churn pool) shows backpressure never fired in any v1 run. The fix is required before any v2 workload that can exhaust the pool.

X. FUTURE WORK

The immediate work is dictated by the limits above. The suite must be rerun on Linux (via OrbStack or a native host) to test whether system malloc's churn-touch strength is a macOS artifact, and LOCUS-EVAL v2 must make the three trace workloads touch-symmetric so the untouched overstatement is priced per workload rather than only in churn form. The backpressure drain-count bug is fixed-before-v2 work. Beyond the suite, the NUMA locality experiment is fully specified and waiting only on hardware: on a multi-node Linux host, two mapped pools bound to different nodes, owner threads pinned per node, and chunk mailboxes steering frees home, measuring cross-node versus node-local write touch on the mixed-lifetime trace. Finally, the design is meant to be embedded: LOCUS ships as the crates.io crate `locus-alloc` v0.1.0 (import name `locus_alloc`) [2], and the open end-to-end question is its integration into a CPU inference engine, where the allocator win must be re-measured as a fraction of real step time that includes KV writes and attention compute.

XI. CONCLUSION

LOCUS makes one claim, and the evidence supports exactly that claim. As a fixed-size KV-block pool for CPU LLM serving, a per-worker lock-free chunk mailbox that the pool owner drains off the allocation hot path is faster than jemalloc, mimalloc, and system malloc on realistic concurrent KV churn, and it is faster while carrying zero tuning parameters, having been reached by deleting a capacity-retune surface rather than building one. At honest touch parity that advantage is 1.6x to 2.7x over mimalloc on the three trace workloads and compresses to

about 1.15x over system malloc once full KV writes dominate, with a transient burst-cancellation footprint of up to 1.5x the theoretical peak as the stated cost of the design. What is not yet proven is stated with equal force: no NUMA-locality speedup is measured, no end-to-end serving result is claimed, and the numbers come from one macOS host on synthetic traces. Those are the next experiments, not the current findings. The value delivered here is a small, parameterless, evidence-backed design and a research record, including its falsified predictions and its self-corrected headline, complete enough for a reader to check every number against a committed source.

REFERENCES

[1] V. Bona, "Locus: NUMA-aware memory pooling primitives for CPU LLM inference serving," GitHub repository, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[2] V. Bona, "locus-alloc, version 0.1.0," crates.io, 2026. [Online]. Available: https://crates.io/crates/locus-alloc

[3] V. Bona, "LOCUS-EVAL v1: standing allocator scoreboard," `documentation/evaluations/0001-locus-eval-v1.md`, Locus research record, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[4] V. Bona, "The Locus research record: experiments 0351 to 0360 and reader's guide," `documentation/README.md` and `documentation/experiments/`, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[5] V. Bona, "ADR 0001: use explicit domain allocators," `documentation/adr/0001-explicit-domain-runtime.md`, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[6] V. Bona, "ADR 0002: narrow system unsafe boundary," `documentation/adr/0002-narrow-sys-unsafe-boundary.md`, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[7] V. Bona, "Serving-engine embedding example," `crates/locus-alloc/examples/serving_engine.rs`, 2026. [Online]. Available: https://github.com/vicotrbb/locus

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proc. 29th ACM Symp. Operating Systems Principles (SOSP)*, 2023. [Online]. Available: https://arxiv.org/abs/2309.06180

[9] J. Evans, "jemalloc: a scalable concurrent malloc implementation," project documentation, 2026. [Online]. Available: https://jemalloc.net/

[10] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: free list sharding in action," Microsoft Research Tech. Rep. MSR-TR-2019-18, 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/

[11] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *Proc. USENIX Summer Tech. Conf.*, 1994. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/bos94/bonwick.html