Drainability: When Coarse-Grained Memory Reclamation Produces Bounded Retention

Dayna Blackwell

2026

Abstract

Long-running services built on coarse-grained, non-relocating memory allocators—slab allocators, region allocators, arena allocators, epoch-based reclamation systems—face a fundamental question: under what conditions is memory retention bounded? We define *drainability* as a structural property of allocation lifetimes relative to the lifecycle of the granule to which they are routed, and prove that it is the necessary and sufficient condition for bounded retention in these systems.

We establish two results:

Alignment Theorem. A granule is reclaimable at its designated reclaim boundary if and only if every allocation routed to it has completed by that boundary.

Pinning Growth Theorem. If a positive fraction of granules retain at least one live allocation at their reclaim boundary, and granules close at a sustained rate, then the number of retained granules grows at least linearly in the number of reclamation cycles.

Together, these results establish a sharp dichotomy: when the routing function aligns allocation lifetimes with granule reclaim boundaries (drainability satisfied), retained memory is bounded by a constant independent of total allocation volume, request count, or uptime. When it does not (drainability violated), retained memory grows without bound. There is no intermediate asymptotic regime—the system either plateaus or diverges, determined entirely by whether the routing function satisfies drainability.

This yields a precise distinction between two failure modes: *logical leaks*, where an allocation is never freed (a correctness bug), and *structural leaks*, where all allocations are eventually freed but lifetime–granularity misalignment prevents granules from draining—producing unbounded memory growth that is invisible to conventional leak detectors.

We validate both results empirically in an epoch-based allocator. A two-point experiment observes a 238× differential in recycle rate between lifetime-mixed and lifetime-isolated routing under identical allocator logic, differing only in allocation routing. A parametric sweep across seven violation fractions ($p = 0$ to $p = 1.0$) confirms that RSS growth scales linearly with the violation fraction ($R^2 \geq 0.998$), with a constant per-epoch cost coefficient, validating the quantitative prediction of the pinning growth theorem. This work does not introduce a new allocator; it characterizes the structural condition that determines whether coarse-grained reclamation produces bounded or unbounded memory retention in long-running systems.

1 Introduction

Consider a long-running service—an HTTP server, a database engine, a message broker—built on a slab or arena allocator. The system passes Valgrind cleanly: every allocation has a matching deallocation. AddressSanitizer reports no errors. Heap profiles show stable live-object counts. Yet under sustained production load, RSS grows linearly and without bound. The team tunes slab sizes, flushes caches more aggressively, increases free-list capacity. Nothing changes the growth slope. The system is eventually killed by the OOM reaper or requires periodic restarts.

This failure mode is not rare. It is a recurring, structurally predictable consequence of how allocations are *routed* to reclamation units—and it is invisible to every standard diagnostic tool because those tools operate at the object level, while the failure exists at the *granule* level.

Coarse-grained memory reclamation is fundamental to systems programming. Slab allocators, region-based allocators, arena allocators, epoch-based reclamation schemes, and non-compacting generational collectors all share a common structure: memory is organized into reclamation units (which we call *granules*), and a granule can be reclaimed only when it contains no live allocations. Despite the ubiquity of this pattern, there is no standard framework for reasoning about when it succeeds. The usual diagnosis for the failure described above is "fragmentation," but this term conflates several distinct phenomena and offers no structural guidance for remediation.

This paper isolates the core structural condition: *drainability.* A granule is drainable if every allocation routed to it completes by the granule's reclaim boundary. We prove that drainability is both necessary and sufficient for bounded memory retention in non-relocating, coarse-grained allocators. When the routing function satisfies drainability, retained memory is bounded by a constant independent of total allocation volume, request count, or uptime—the system plateaus. When it is violated, retained memory grows at least linearly—the system diverges. There is no intermediate asymptotic regime. This sharp dichotomy is determined entirely by the routing function, not by allocator policy, tuning, or heuristics.

The violation case produces what we call a *structural leak*: a class of memory growth failure that is distinct from logical leaks (forgotten deallocations), that is invisible to object-level tools, and that cannot be remediated by allocator tuning. The only fix is architectural: route allocations to granules whose reclaim boundaries match allocation lifetimes.

The contribution is one of framing and proof. The alignment theorem is a direct consequence of the definitions, which is a feature rather than a limitation: it means the abstraction is well-chosen. The value lies in proving the bounded/unbounded dichotomy, naming the condition that determines which regime a system occupies, distinguishing structural leaks from logical leaks, and demonstrating that allocation *routing*—not allocator *policy*—is the determining factor. For practitioners building long-running services on coarse-grained allocators, the central result is concrete: lifetime-aligned routing is the necessary and sufficient architectural condition for bounded memory retention.

Scope. The drainability framework applies to allocators that reclaim memory at granularity boundaries without relocating live objects across granules. Compacting collectors, which can consolidate live objects into fewer granules through relocation, alter the reclaimability condition fundamentally and are outside the scope of this analysis. We discuss the relationship to compacting collection in Section 7.

2 Model

2.1 Allocations and Granules

Let $\mathbf{A}$ be the set of allocations and $\mathbf{G}$ be the set of reclamation units (*granules*: slabs, regions, generations, epochs).

Terminology. We use *granule* as a unifying term for any coarse-grained reclamation unit—a contiguous block of memory that the allocator treats as an atomic unit for reclamation purposes. This encompasses slabs [1], regions and arenas [9], epochs [3], and non-compacting generations. We adopt a neutral term deliberately: the theorems in this paper apply to all of these mechanisms, and using any implementation-specific name would incorrectly suggest a narrower scope.

Each allocation $a \in \mathbf{A}$ has a lifetime:

$$\mathit{lifetime}(a) = [t_{\mathit{alloc}}(a),\ t_{\mathit{free}}(a)]$$

where $t_{\mathit{free}}(a) < \infty$ for all allocations we consider (the case $t_{\mathit{free}}(a) = \infty$ is a logical leak, treated separately in Section 4).

Each granule $g \in \mathbf{G}$ has a lifecycle with a designated reclaim boundary:

$$\mathit{lifecycle}(g) = [t_{\mathit{open}}(g),\ t_{\mathit{reclaim}}(g)]$$

Each allocation is routed to exactly one granule by a routing function:

$$\rho : \mathbf{A} \to \mathbf{G}$$

We write $\rho(a)$ for the granule to which allocation a is routed. We write $\mathit{live}(g,t) = \{a \in \mathit{allocs}(g) : t_{\mathit{free}}(a) > t\}$ for the set of allocations in g that are live at time t. Drainability (defined in Section 3) is a property of ρ given the program's lifetime structure: the same program may produce drainable granules under one routing function and non-drainable granules under another. The central result of this paper is that reclamation success is determined by ρ, not by the allocator's reclamation policy.

2.2 Reclamation Rule

We consider *non-relocating* reclamation: a granule g is reclaimable at time t if and only if it contains no live allocations at t:

$$\mathit{Reclaimable}(g,t) \iff \forall a:\ \rho(a) = g \implies t \geq t_{\mathit{free}}(a)$$

Equivalently:

$$\mathit{Reclaimable}(g,t) \iff \mathit{live}(g,t) = \emptyset$$

This captures slab reclamation, region deallocation, non-compacting generation collection, and epoch close. The key constraint is that live objects are never moved between granules.

Assumption 1 (Non-Relocating Retention)**.** We assume that a granule that is not reclaimable at its designated boundary remains retained until it becomes reclaimable at some later time. In particular, the allocator does not relocate live objects across granules, and granules are not merged across reclaim boundaries. This assumption connects the alignment theorem (Section 3.2) to the growth bounds (Section 5): a non-drainable granule at its boundary necessarily contributes to retained memory.

3 Drainability

3.1 Definition

A granule g is **drainable** if and only if every allocation routed to g completes by the reclaim boundary. Writing $\mathit{allocs}(g) = \{a \in \mathbf{A} : \rho(a) = g\}$ for the set of allocations routed to g:

$$\mathit{Drainable}(g) \iff \forall a \in \mathit{allocs}(g) : \; t_{\mathit{free}}(a) \leq t_{\mathit{reclaim}}(g)$$

Drainability is not purely an allocator property; it is a joint property of (1) the program's allocation lifetime structure and (2) the routing function ρ. The same program may produce drainable granules under one routing and non-drainable granules under another. This is the central observation of the paper.

3.2 Alignment Theorem

Theorem 1 (Reclaimability $\iff$ Drainability at the Boundary)**.** *For any non-relocating, coarse-grained allocator, a granule g is reclaimable at its designated reclaim boundary $t_{\mathit{reclaim}}(g)$ if and only if g is drainable.*

Proof. ($\Rightarrow$) Suppose g is reclaimable at $t_{\mathit{reclaim}}(g)$. Then $\mathit{live}(g, t_{\mathit{reclaim}}(g)) = \emptyset$, so for every $a \in \mathit{allocs}(g)$, we have $t_{\mathit{free}}(a) \leq t_{\mathit{reclaim}}(g)$. Hence g is drainable.

($\Leftarrow$) Suppose g is drainable. Then for every $a \in \mathit{allocs}(g)$, we have $t_{\mathit{free}}(a) \leq t_{\mathit{reclaim}}(g)$, so $\mathit{live}(g, t_{\mathit{reclaim}}(g)) = \emptyset$, and g is reclaimable at $t_{\mathit{reclaim}}(g)$. □

The biconditional follows directly from the definitions. This is by design: the purpose of the drainability abstraction is to name the exact condition that separates reclaimable from non-reclaimable granules, not to derive a surprising consequence.

The practical content of the theorem is that non-drainable granules induce *pinning pressure*: a single long-lived allocation can prevent reclaiming the entire granule and all the memory it contains.

4 Failure Modes

Two structurally distinct failure modes produce retained memory growth:

Logical leak (object-level).

$$\exists a:\ t_{\mathit{free}}(a) = \infty$$

An allocation is never freed. This is a correctness bug, detectable by tools such as Valgrind and AddressSanitizer.

Structural leak (granule-level).

$$\forall a:\ t_{\mathit{free}}(a) < \infty, \quad \text{but} \quad \exists g:\ \mathit{live}(g, t_{\mathit{reclaim}}(g)) \neq \emptyset$$

Every allocation is eventually freed, but lifetime mixing prevents granules from draining at their reclaim boundaries.

The structural leak is the more insidious failure mode precisely because it is invisible to object-level leak detectors. A system exhibiting structural leaks will pass Valgrind cleanly: every allocation has a corresponding deallocation. Yet RSS grows without bound because granules accumulate faster than they drain.

Structural Leak Principle. In granularity-based reclamation systems, retained-granule growth (and therefore retained memory under non-relocating retention) may be asymptotically forced by lifetime–granularity misalignment, even in the complete absence of logical leaks.

The practical consequence of this distinction is that each failure mode demands a different remediation strategy:

Diagnosis	**Cause**	**Remediation**
Logical leak	Missing deallocation	Find the unmatched `alloc`
"Fragmentation" (misdiagnosis)	Assumed allocator inefficiency	Tune slab sizes, flush caches (cannot change asymptotic class)
Structural leak	Lifetime–granularity misalignment in ρ	Change the routing function ρ

The third row is the actionable contribution of the drainability framework. Systems exhibiting structural leaks are routinely misdiagnosed as the second row, leading to remediation efforts that cannot succeed.

5 Growth Bounds

We characterize retained memory growth under drainability satisfaction versus violation.

5.1 Slab-Granularity Refinement

For slab allocators with size classes, let: k denote a size class, C_k the objects per slab (capacity for class k), $L_k(g, t)$ the number of live objects of class k in granule g at time t, and $R_k(g, t) = \lceil L_k(g, t)/C_k \rceil$ the retained slabs of class k in g at time t.

A slab is reclaimable if and only if it contains zero live objects, so any live object pins its entire slab. Total retained slabs at time t (slabs in closed granules that are not yet reclaimable):

$$R(t) = \sum_{\substack{g \text{ closed by } t \\ \mathit{live}(g,t) \neq \emptyset}} \sum_k R_k(g,t)$$

5.2 Assumptions

We make three bounding assumptions to isolate structural effects from allocator-internal behaviors:

1. **Bounded recycling caches.** The number of empty slabs held in free lists or recycling caches is bounded by a constant K.
2. **Bounded pipeline depth.** The number of simultaneously open granules is bounded by a constant G_{open}.
3. **Non-merging.** Granules are not merged across reclaim boundaries. Each violating granule contributes a distinct set of retained slabs.

These assumptions are satisfied by practical allocators and ensure that any unbounded growth in $R(t)$ is attributable to structural pinning rather than unbounded internal caching.

5.3 Bounded Growth Under Drainability (Theorem 2)

Theorem 2. *If every closed granule is drainable, then $R(t)$ is bounded independent of time.*

Proof. Drainability implies $L_k(g,t) = 0$ for all size classes k, all closed granules g, and all $t \geq t_{reclaim}(g)$, so every closed granule satisfies $\mathit{live}(g,t) = \emptyset$ and is excluded from the sum. Retained slabs therefore consist only of slabs in currently open granules and slabs in bounded recycling caches. Let peak_k be the maximum number of live objects of class k in any single open granule. Then:

$$R(t) \leq G_{open} \cdot \sum_k \lceil \mathit{peak}_k / C_k \rceil + K$$

This bound is constant in t. □

Corollary 1. *Under drainability, RSS converges to a steady-state plateau determined by the pipeline depth, peak per-granule allocation density, and cache capacity—not by total allocation volume.*

5.4 Linear Lower Bound Under Violation (Theorem 3)

Theorem 3. *Let $m(t)$ be the number of granules closed by time t. If there exists $p > 0$ such that for at least a fraction p of closed granules, at time t there exists some size class k with $L_k(g,t) \geq 1$, then:*

$$R(t) \geq p \cdot m(t)$$

Proof. Each such granule contributes at least one retained slab at time t (since $\lceil L_k(g,t)/C_k \rceil \geq 1$ when $L_k(g,t) \geq 1$). With at least $p \cdot m(t)$ such granules by time t, the bound follows. $\square$

Instantiation for batch workloads. For workloads processing requests of average size B allocations, granule closes scale as $m(t) \approx t/B$, yielding $R(t) = \Omega(t/B)$.

Corollary 2. *Under sustained drainability violation, RSS grows without bound, at a rate proportional to the request processing rate.*

On "sustained" violation. Theorem 3 requires that at any observation time t, a positive fraction p of granules closed by t still contain at least one live allocation. This holds when pinning allocations' lifetimes span many granule closings—the common case for server workloads where long-lived objects (sessions, connections, caches) are mixed with short-lived objects (requests). Under steady-state workload and stable routing, new granules are pinned at a constant rate while old pinning allocations remain live, so the fraction of currently-pinned granules remains at least p and $R(t)$ grows linearly in $m(t)$. Transient mixing—a brief burst followed by clean routing—produces a bounded one-time cost rather than unbounded growth, because the pinning allocations are eventually freed and the affected granules become reclaimable.

5.5 The Bounded/Unbounded Dichotomy

Theorems 2 and 3 together establish a sharp asymptotic dichotomy for non-relocating, coarse-grained allocators under sustained workload:

- **Drainability satisfied** (ρ aligns lifetimes with reclaim boundaries): $R(t) = O(1)$. Retained memory is bounded by a constant independent of total allocation volume, request count, or uptime. The system can run indefinitely without memory growth.
- **Drainability violated** (a positive fraction of granules are non-drainable): $R(t) = \Omega(t)$. Retained memory grows at least linearly. No allocator tuning can convert this to bounded growth within the non-relocating model.

There is no intermediate asymptotic regime. The routing function ρ determines which regime the system occupies. This is the central result of the paper: for long-running services built on coarse-grained allocators, lifetime-aligned routing is the necessary and sufficient condition for bounded memory retention.

6 Empirical Validation

We validate the alignment and pinning growth theorems using an epoch-based allocator with controlled routing.

6.1 Two-Point Dichotomy

An initial experiment established the qualitative dichotomy. A slab-granularity allocator processed identical workloads (session and request objects) under two routing functions: one mixing lifetime classes within granules (drainability violated), one isolating them (drainability satisfied). All allocator parameters were held constant; only routing changed.

Metric	Mixed (Violation)	Isolated (Satisfaction)	Predicted
Recycle rate	0.28%	66.5%	—
Retained slabs	~493K	~2K	$\Omega(t)$ vs $O(1)$
RSS behavior	Linear growth	Plateau	$\Omega(t)$ vs $O(1)$

The recycle-rate differential is 238× (66.5% / 0.28%). This confirms the qualitative prediction: drainability satisfaction produces bounded retention, violation produces unbounded growth. However, two data points cannot test the quantitative relationship between the violation fraction and the growth rate. The following experiment addresses this.

6.2 Parametric Validation: P-Sweep

To validate Theorem 3 quantitatively, we vary the fraction p of granules that receive a cross-lifetime allocation and measure the resulting RSS growth.

Design. An epoch-based allocator processes 200K requests with epoch advances every 2 requests, producing exactly 100K epoch closes ($m = 100{,}000$). At each epoch open, a Bernoulli trial with probability p determines whether a single session-scoped object is allocated into that epoch. Session objects are never freed during the run; request objects are freed at epoch close. The allocator implementation, slab sizes, cache policy, object sizes, and request count are held constant across all seven runs. The only variable is p.

Results.

p	Sessions	RSS Growth	R^2	Behavior
0.00	0	0.12 MB	—	$O(1)$ plateau
0.01	1,014	0.50 MB	0.905	Linear (noisy)
0.05	4,924	2.12 MB	0.995	Linear
0.10	10,202	4.25 MB	0.998	Linear
0.25	24,909	10.38 MB	1.000	Linear
0.50	49,965	20.50 MB	1.000	Linear
1.00	99,999	41.00 MB	1.000	Linear

Figure 1 shows RSS over time for all seven runs. At $p = 0$, RSS is constant at 1.50 MB for the duration of the run. For all $p > 0$, RSS grows linearly and without bound, with slope proportional to p.

Linearity in p. Dividing RSS growth by p yields a nearly constant per-epoch memory cost:

For $p \geq 0.05$, the ratio is within 4% of 41.5 MB per unit p, confirming that $R(t) \approx c \cdot p \cdot m(t)$ for a configuration-dependent constant c (the average bytes retained per pinned epoch). The $p = 0.01$ outlier reflects Bernoulli variance at low event counts. This linear relationship is tighter than the lower bound predicted by Theorem 3.

Linearity in t. To verify that growth is sustained rather than transient, we measured the slope of $R(t)$ across quartiles of the $p = 1.0$ run. The per-request growth rate was 0.214 KB/req in

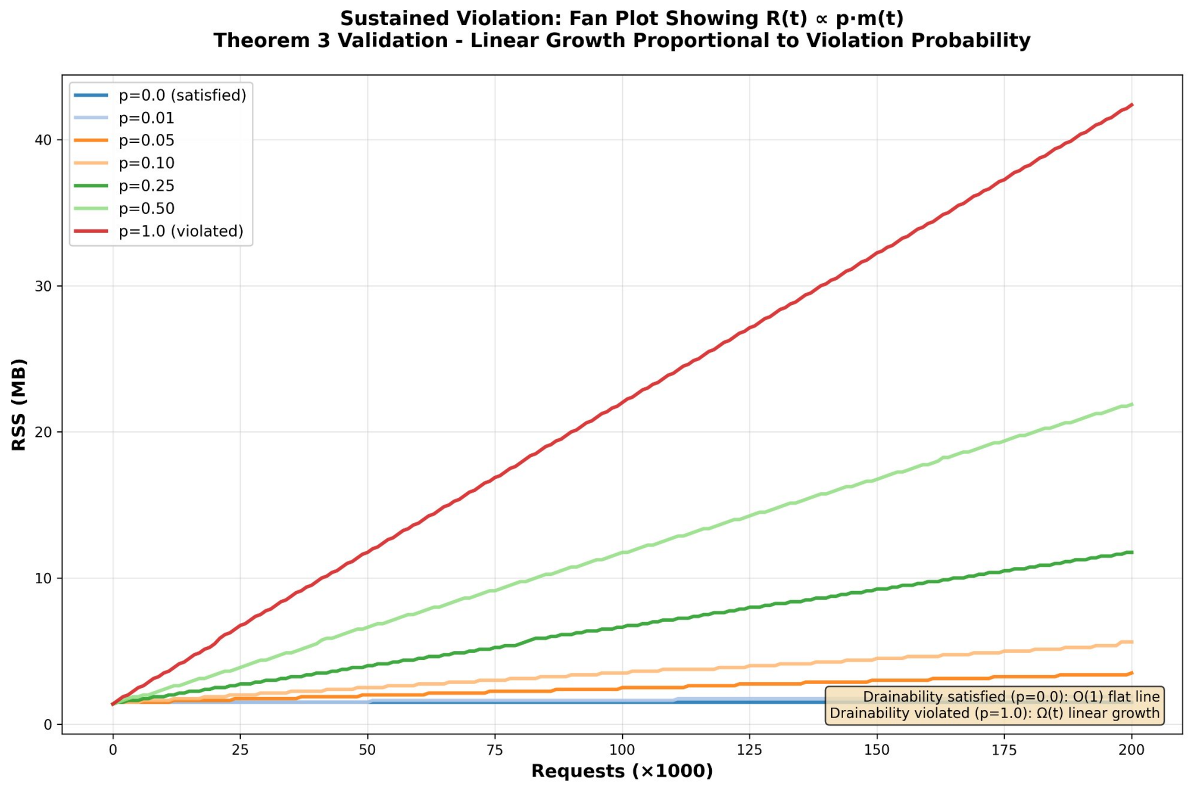

Figure 1: RSS over time for violation probabilities $p \in \{0, 0.01, 0.05, 0.10, 0.25, 0.50, 1.0\}$. Drainability satisfied ($p = 0$): RSS is constant. Drainability violated ($p > 0$): RSS grows linearly with slope proportional to p. All runs use identical allocator logic, slab parameters, and workload; only the routing function varies.

the first quartile, 0.209 in the second, 0.209 in the third, and 0.207 in the fourth—a quartile slope ratio of 0.989, indicating constant growth rate throughout the measurement window. No concavity was detected at any p value ($R^2 \geq 0.998$ for $p \geq 0.05$), confirming that pinned epochs are not being reclaimed during the run.

Detection threshold. At $p = 0.01$, the growth (0.50 MB over 200K requests) is real but noisy ($R^2 = 0.905$). At $p = 0.05$, the signal is unambiguous ($R^2 = 0.995$). This suggests a practical detection threshold: a 1% drainability violation rate produces growth that is measurable with targeted instrumentation, while a 5% rate is visible with standard RSS monitoring.

6.3 Limitations

Both experiments use synthetic workloads with two lifetime classes (session and request objects) and a single allocator implementation. The 238× recycle-rate differential and the ∼41 MB per unit p coefficient are specific to these configurations' parameters. The theoretical results (Theorems 2 and 3) are general, but validating them across multiple lifetime classes, multiple allocator implementations, real application traces, and varying size-class distributions remains future work. The p-sweep validates the predicted relationship between violation fraction and growth rate but does not characterize how the per-epoch cost c varies across allocator configu-

p	Growth / p (MB)
0.01	50.0
0.05	42.4
0.10	42.5
0.25	41.5
0.50	41.0
1.00	41.0

rations.

7 Discussion

7.1 Allocation Routing Is a Structural Decision

The drainability framework reveals that allocation routing is not merely an optimization concern but a structural one. Reclamation success depends on whether the routing function respects lifetime boundaries, not on how frequently granules are closed or how aggressively caches are managed.

A routing-aware allocation pattern separates lifetime classes explicitly:

```
// Long-lived allocations -> persistent scope
session_scope_enter();
allocate_session_object();
session_scope_exit();

// Short-lived allocations -> rotating scope
request_scope_enter();
allocate_request_object();
request_scope_exit();  // granule close triggers reclamation
```

7.2 The Limits of Policy

Under sustained drainability violation, no amount of allocator tuning can restore bounded memory growth within the non-relocating model. More frequent granule closes do not help—they simply create more non-drainable granules faster. Heuristic reordering of allocations within a granule does not help—the pinning object remains.

However, it is worth noting that practical mitigations can reduce the *constant factor* of the growth. Segregated free lists that probabilistically cluster allocations by lifetime, lazy coalescing strategies, and partial compaction within a granule can all slow the rate of growth. The drainability framework predicts that these approaches change the coefficient but not the asymptotic class: under sustained mixing, growth remains $\Omega(m(t))$ regardless of mitigation effort.

The only *asymptotic* fix is architectural: route allocations so that each granule's contents are lifetime-homogeneous with respect to the granule's reclaim boundary.

7.3 Diagnosing and Fixing a Structural Leak: A Worked Example

To make the framework actionable, we walk through a realistic structural leak as it would present in production, diagnosed and resolved using drainability analysis.

System. Consider an HTTP API service using a slab allocator with rotating arenas. Each incoming request opens an arena (granule), allocates objects to service the request, and closes the arena on response completion. The allocator reclaims arenas at close.

The service also maintains a connection pool. When a new client connects, the service allocates a connection metadata object (TLS state, client identity, rate-limit counters). These connection objects are long-lived—they persist for the duration of the client session, which spans hundreds or thousands of requests.

Symptom. After deploying to production, the SRE team observes that RSS grows linearly under sustained load—approximately 12 MB per hour at 500 requests/second. The service passes Valgrind cleanly. AddressSanitizer reports no leaks. All allocations have matching deallocations. Heap profiling shows stable live-object counts. The team suspects "fragmentation" and begins tuning: adjusting slab sizes, increasing free-list capacity, adding more aggressive cache flushing. None of these changes alter the growth slope.

Diagnosis with drainability analysis. The framework directs attention to a specific question: *are there allocations whose lifetimes exceed the reclaim boundary of the granule they are routed to?*

Examining the allocation routing reveals the problem. When accepting a new connection, the connection pool allocates connection metadata using the *current request arena* – whichever arena happens to be active at that moment:

```
fn handle_request(arena: &Arena) {
    if new_connection {
        // Connection metadata allocated in request arena
        let conn = arena.alloc::<ConnMetadata>();  // structural violation
        connection_pool.register(conn);
    }
    let req = arena.alloc::<RequestData>();
    process(req);
}
// arena.close() -- but conn is still live
```

This is a drainability violation. The connection object's lifetime (minutes to hours) exceeds the request arena's reclaim boundary (milliseconds). Every arena that happens to service a new connection is pinned for the duration of that connection. Under steady-state connection churn, a constant fraction of arenas are pinned at close, producing the linear growth predicted by Theorem 3.

Critically, this is not a logical leak. The connection metadata *is* freed—when the client disconnects, the connection object is deallocated. But by then, the request arena it was allocated in has been closed and pinned for the entire session duration, retaining all the memory it contains.

Why conventional tools miss it. Valgrind and ASan track object-level correctness: was every `alloc` matched by a `free`? Yes. Heap profilers track live-object counts: are objects accumulating? No—live counts are stable. RSS monitors detect growth but cannot attribute it to a cause. The failure exists at the *granule* level, in the relationship between allocation routing and lifetime structure, which no object-level tool inspects.

Fix. The drainability framework prescribes the remedy directly: route allocations to granules whose reclaim boundaries match allocation lifetimes. Connection metadata belongs in a connection-scoped granule, not a request-scoped one:

```
fn accept_connection(conn_arena: &Arena) {
    // Connection metadata in connection-scoped arena
    let conn = conn_arena.alloc::<ConnMetadata>();
    connection_pool.register(conn);
}

fn handle_request(req_arena: &Arena) {
    let req = req_arena.alloc::<RequestData>();
    process(req);
}
// req_arena.close() -- fully drainable, no pinning
// conn_arena reclaimed when connection closes
```

After this change, every request arena contains only request-scoped allocations. Drainability is restored: all allocations in each request arena complete before the arena's reclaim boundary. RSS plateaus as predicted by Theorem 2.

What changed. The allocator is identical. The slab sizes are identical. The cache policy is identical. The workload is identical. The only change is one routing decision—which arena a connection object is allocated in. That single routing change is the difference between $\Omega(t)$ and $O(1)$ memory growth.

7.4 Relationship to Compacting Collection

Compacting garbage collectors (and relocating generational collectors) operate under a fundamentally different reclamation rule: live objects can be *moved* from one granule to another, consolidating surviving objects into fewer granules and freeing the rest. This violates the non-relocating assumption of Section 2.2 and changes the reclaimability condition entirely.

In a compacting collector, a granule need not be drainable to be reclaimed—the collector simply evacuates its live objects first. Generational collectors exploit this directly: objects that survive a young-generation collection are *promoted* (relocated) to an older generation, allowing the young generation to be reclaimed even though it contained long-lived objects.

Drainability is therefore *not* a necessary condition for reclamation in general—it is a necessary condition for reclamation *in non-relocating systems*. The framework characterizes exactly the systems where architectural routing discipline is the only path to bounded memory, as opposed to systems where the runtime can compensate through relocation. This distinction helps practitioners choose between allocator designs: if relocation is acceptable (e.g., in a managed runtime), compaction can mask lifetime mixing. If relocation is not acceptable (e.g., in systems with raw pointers, foreign function interfaces, or real-time constraints), drainability becomes a hard requirement.

7.5 Relationship to Fragmentation

"Fragmentation" is commonly used to describe the symptom that drainability violation produces, but the term conflates several distinct phenomena: *internal* fragmentation (wasted space within an allocated block), *external* fragmentation (free memory that cannot satisfy a request due to non-contiguity), and what we call *structural* fragmentation (granules that cannot be reclaimed due to pinning). The drainability framework provides a precise characterization of the third category and separates it from the first two, which have different causes and different remedies.

8 Related Work

The research lineages below each address a specific mechanism, type system, or bound for memory management. None states the structural condition that governs reclamation success across all coarse-grained, non-relocating systems. The drainability framework fills that gap.

8.1 Granularity-Based Systems

The drainability condition applies uniformly to non-relocating systems that reclaim at granularity boundaries, including: slab allocators [1] (Linux SLUB), where the granule is a slab and the reclaim boundary is when all objects are freed; region and arena allocators [9] (Rust's `bumpalo`), where the granule is a region and the reclaim boundary is region deallocation; epoch-based reclamation [3] (crossbeam), where the granule is an epoch and the reclaim boundary is epoch quiescence; and non-compacting generational schemes, where the granule is a generation and the reclaim boundary is generation collection without relocation.

Among these systems, epoch-based reclamation (EBR) merits particular attention because it is where drainability violations most commonly arise in practice, albeit under different terminology. In EBR systems such as Crossbeam (Rust) or RCU (Linux), threads enter an epoch by acquiring a guard, and objects retired in that epoch cannot be reclaimed until all guards from that epoch have been released. The well-known failure mode—a slow or stalled reader holding a guard past an epoch boundary, thereby preventing reclamation of all objects retired in that epoch—is precisely a drainability violation: the guard's lifetime exceeds the reclaim boundary of its granule (the epoch). Practitioners describe this as "epoch pinning" or "stalled reclamation," and the standard remediation (ensuring readers do not hold guards across epoch boundaries) is exactly what the drainability framework prescribes: align the lifetime of each allocation (the guard) with the reclaim boundary of its granule (the epoch). Drainability thus generalizes this EBR-specific failure mode into a uniform condition that applies across all coarse-grained reclamation systems.

In each case, the alignment theorem applies: deterministic reclamation at the boundary requires drainability of the granule. Bonwick [1] provided the mechanism for efficient slab allocation; the drainability framework provides the structural precondition for its long-term sustainability under workload churn. Similarly, Tofte and Talpin [9] established region-based lifetime management as a compiler-directed strategy; drainability characterizes when that strategy succeeds at the reclamation level regardless of whether lifetime inference is compiler-directed or programmer-directed.

It is worth noting that Tofte and Talpin also identified *region proliferation*—the creation of excessive regions when lifetime inference cannot prove that objects in different regions share lifetimes—as a failure mode of region-based management. Region proliferation is the dual of drainability violation: where drainability violation places too many lifetime classes in too few granules (causing pinning), region proliferation places too few objects in too many granules (causing overhead from excessive region creation and bookkeeping). Both are failures of lifetime–granule alignment, but in opposite directions.

8.2 Randomized Compaction and Physical Page Reclamation

Mesh [2] addresses a closely related problem—reclaiming physical memory from partially-occupied allocator spans—through a technique the authors call *meshing*: randomly assigning allocations to virtual pages such that, with high probability, pairs of partially-occupied pages can be overlaid onto a single physical page without relocating objects. Mesh operates within our framework's scope (non-relocating at the virtual level) but circumvents the drainability requirement at the physical level by exploiting virtual memory aliasing. In drainability terms, Mesh does not make granules drainable; it instead reduces the physical memory cost of non-drainable granules by merging their non-overlapping contents. This is a complementary strategy: Mesh mitigates the *constant factor* of pinning-induced retention, while the drainability framework characterizes the *asymptotic class*. Under sustained lifetime mixing, Mesh reduces the coefficient of the linear growth but does not convert it to $O(1)$—that requires routing discipline.

mimalloc [6] organizes memory into segments divided into pages, with per-thread free lists and a segment reclamation strategy: a segment is reclaimed when all of its pages are empty. This is a direct instance of two-level granularity-based reclamation, and the drainability condition applies at both levels—a page is reclaimable when all its objects are freed, and a segment is reclaimable when all its pages are reclaimed. mimalloc's design mitigates cross-thread pinning through thread-local allocation, which is a form of implicit lifetime routing (thread-local objects tend to share lifetime characteristics). The drainability framework explains *why* this design choice is effective: thread-local routing probabilistically increases drainability by reducing lifetime mixing within each segment.

8.3 Fragmentation Theory

Robson [7, 8] established worst-case lower bounds on memory fragmentation for any allocator: for allocations of sizes in $\{1, \ldots, m\}$ with maximum live memory M, any allocator may require $\Omega(M \cdot \log m)$ total memory in the worst case. These bounds characterize *external* fragmentation—the inability to satisfy allocation requests despite sufficient total free memory. The drainability framework addresses a different phenomenon: *structural* fragmentation, where free memory exists within granules but cannot be reclaimed because the granule also contains live objects. Robson's bounds apply to the allocation-level question ("can this request be satisfied?"), while drainability addresses the reclamation-level question ("can this granule be returned to the system?"). The two are complementary: a system can satisfy Robson's bounds (no allocation failures) while violating drainability (unbounded granule retention), and vice versa. Johnstone and Wilson [5] showed empirically that practical fragmentation is far below Robson's worst case for most workloads; the drainability framework similarly predicts that practical retention is bounded when routing respects lifetime structure, and unbounded when it does not.

9 Future Work

Static analysis of drainability. Can drainability be checked at compile time? In the general case, determining whether an allocation's lifetime exceeds its granule's reclaim boundary requires resolving control flow and input-dependent behavior, making exact static decidability unlikely for arbitrary programs. However, conservative approximations are feasible. Rust's ownership and borrowing system already enforces a restricted form of lifetime-scoped allocation—an allocation borrowed by a scope cannot outlive that scope, which is a syntactic sufficient condition for drainability when scopes correspond to granules. Region type systems [9, 4] enforce similar constraints through type-level region annotations. Investigating what class of routing functions can be statically verified as drainability-preserving—and what the precision-cost tradeoff looks like for practical codebases—is an open question.

Dynamic detection. A more immediately tractable direction is runtime detection of drainability violations. We envision a *drainability profiler* operating in two modes. A *production mode* would record one bit per granule close—drainable or not—and expose the drainability satisfaction rate as a continuously monitored metric, analogous to a cache hit rate. The overhead is minimal (one timestamp comparison per close), making it viable for always-on deployment. A drop in the drainability rate would signal a structural regression before RSS growth becomes visible.

A *diagnostic mode*, enabled during investigation, would additionally record the allocation site and timestamp of every object live at granule close, together with the granule's reclaim timestamp. The difference between allocation lifetime and reclaim boundary—the *lifetime delta*—distinguishes near-miss violations (an object freed milliseconds after close, potentially addressable by adjusting granule boundaries) from fundamental routing mismatches (an object living hours in a millisecond-scoped granule, requiring an architectural routing change). Reporting the pinning allocation site directly identifies the line of code responsible for the structural leak.

No such tool currently exists. The diagnostic gap is precisely the one identified in Section 7.3: conventional tools operate at the object level and cannot attribute RSS growth to granule-level retention. A drainability profiler would close this gap by measuring the necessary and sufficient condition for reclamation success, rather than a proxy. Integration as a reporting mode in existing allocators (jemalloc, mimalloc, bumpalo) would provide the most direct path to adoption.

Empirical breadth. The p-sweep validates the predicted relationship between violation fraction and growth rate for a single allocator and workload. Extending this to multiple allocator implementations (slab, region, epoch), three or more lifetime classes, real application traces, and sensitivity to granule parameters (slab capacity, region size) would strengthen the empirical case further. Of particular interest is characterizing how the per-epoch cost coefficient c varies across allocator configurations, and whether the practical detection threshold ($p \approx 0.01$–0.05 in our experiments) is stable across workloads.

Routing function design. This paper characterizes when a routing function ρ produces drainable granules but does not address how to design ρ. Formalizing sufficient conditions on routing functions—for instance, temporal clustering strategies that group allocations by predicted lifetime class—would extend the framework from diagnosis to synthesis. The relationship between lifetime prediction accuracy and drainability satisfaction rates is unexplored.

Generalization across allocator hierarchies. The drainability condition applies at any granularity boundary in a non-relocating system, including virtual memory pages in general-purpose allocators. The practical severity of a drainability violation, however, depends on whether the pinned granule permits internal reuse of freed space: arena-style granules waste the entire pinned region, while slab and page-level granules may recycle free slots internally, reducing the memory cost per pinned granule without eliminating the retention itself. Characterizing this severity spectrum—and the interaction between drainability violations at multiple levels of an allocator hierarchy—is a natural extension of the present framework.

10 Conclusion

We establish two structural results for coarse-grained, non-relocating memory reclamation:

Alignment Theorem:

$$\mathit{Reclaimable}(g, t_{\mathit{reclaim}}(g)) \iff \forall a \in \mathit{allocs}(g) : \; t_{\mathit{free}}(a) \leq t_{\mathit{reclaim}}(g)$$

Pinning Growth Theorem:

$$\text{Under sustained violation with fraction } p > 0 : \quad R(t) \geq p \cdot m(t)$$

Empirically, a two-point experiment demonstrates the qualitative dichotomy: a 238× recycle-rate differential (0.28% vs 66.5%) under identical allocator logic, differing only in allocation routing. A parametric p-sweep across seven violation fractions ($p = 0$ to $p = 1.0$) validates the quantitative prediction: RSS growth scales linearly with p ($R^2 \geq 0.998$ for $p \geq 0.05$), with a constant per-epoch cost coefficient stable within 4% across two orders of magnitude of the violation fraction.

Core result. For long-running services built on non-relocating, coarse-grained allocators, drainability is the necessary and sufficient condition for bounded memory retention. When the routing function ρ aligns allocation lifetimes with granule reclaim boundaries, retained memory is bounded by a constant independent of total allocation volume, request count, or uptime (Theorem 2). When it does not, retained memory grows at least linearly (Theorem 3). This is a sharp dichotomy with no intermediate asymptotic regime: the system either plateaus or diverges, and the outcome is determined entirely by the structural alignment between allocation routing and lifetime semantics—not by allocator policy, tuning, or heuristics.

The drainability framework provides both the diagnostic question ("does any allocation outlive its granule's reclaim boundary?") and the architectural prescription ("route to a granule whose boundary matches the allocation's lifetime"). It explains a class of memory growth failures—structural leaks—that are invisible to conventional leak detection, and it defines the conditions under which coarse-grained reclamation can sustain bounded memory in perpetuity.

References

[1] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proc. USENIX Summer 1994 Technical Conference*, pp. 87–98, 1994.

[2] B. Emery, E. D. Berger, and C. Curtsinger. Mesh: Compacting memory management for C/C++ applications. In *Proc. 40th ACM SIGPLAN PLDI*, pp. 333–346, 2019.

[3] K. Fraser. *Practical Lock-Freedom.* PhD dissertation, University of Cambridge, 2004.

[4] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, pp. 275–288, 2002.

[5] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proc. ACM SIGPLAN ISMM*, pp. 26–36, 1998.

[6] D. Leijen, B. Zorn, and L. de Moura. mimalloc: Free list sharding in action. In *Proc. APLAS*, pp. 244–265, 2019.

[7] J. M. Robson. An estimate of the store size required for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, 1971.

[8] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[9] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
