Garbage Collection for Concurrent Data Structures in Rust

The Alloy Approach

Xinyu Zheng Antony Hosking Eduardo Amorim

PLISS 2026

Australian National University

Introduction & Motivation

What This Talk Is About

Core question: Can a tracing GC simplify safe memory reclamation in lock-free concurrent data structures?

Approach: *Alloy* (Hughes and Tratt 2025) — a fork of the Rust compiler that adds a first-class `Gc<T>` type backed by the Boehm–Demers–Weiser (BDW) collector.

We study: two production-quality concurrent maps (papaya, SCC) and how their reclamation logic simplifies under Alloy.

Roadmap

1. Memory management in concurrent code
2. Techniques: epoch-based reclamation (EBR), hazard pointers, Hyaline
3. DashMap, papaya, SCC as case studies
4. Alloy: `Gc<T>` in Rust
5. Our hybrid allocation strategy
6. Status & discussion

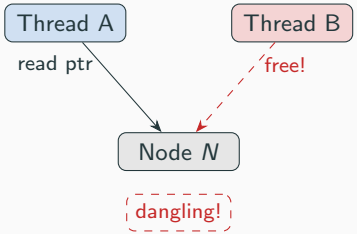
The Problem: Reclaiming Memory Concurrently

Sequential reclamation is easy:

- `Box<T>` drops when no longer reachable
- Rust borrow checker enforces single-owner discipline

Concurrent reclamation is *hard*:

- Thread A reads a pointer to node N
- Thread B removes and frees N
- Thread A dereferences dangling pointer **(use-after-free)**



The fundamental challenge: we cannot free a node while *any* thread might still hold a reference to it.

A Spectrum of Solutions

Technique	Overhead	Blocking?	Cycles?
Stop-the-world	high latency	yes	yes
`Arc<T>`	CAS per clone	no	no
Hazard pointers	scan overhead	no	yes
Epoch-based (EBR)	low avg.	slow threads	yes
Hyaline (seize)	low avg.	no	yes
Tracing GC	GC pauses	occasional	yes

All of these appear in practice. Today we trace the line from EBR/Hyaline (as used in *papaya* and *SCC*) to tracing GC (Alloy).

Memory Reclamation Techniques

Reference Counting: `Arc<T>`

Idea: each object carries a reference count. Free when count reaches zero.

Rust: `Arc<T>` (Atomically Reference Counted)

- Clone: atomic increment
- Drop: atomic decrement; free if zero
- No extra synchronisation per read

Weaknesses:

- CAS on every clone/drop — cache contention
- Cannot collect *cycles*
- High overhead for short-lived values

```rust
struct Node {
    value: i64,
    next: Option<Arc<Node>>,
}

// safe, but every clone is a CAS
let a = Arc::new(Node { ... });
let b = Arc::clone(&a); // +1
drop(b);                // -1 (CAS)
// free when count == 0
```

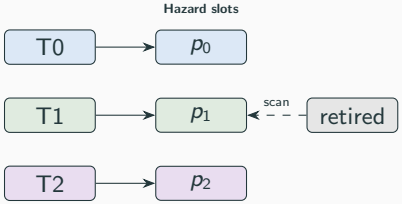
Hazard Pointers

Idea *(Michael 2004)*: before dereferencing a shared pointer, a thread publishes it in a *hazard record* (per-thread array of "protected" pointers).

Protocol:

1. Read pointer p from shared structure
2. Store p in local hazard slot
3. Re-read: if p changed, retry
4. Dereference p safely
5. Clear hazard slot when done

Reclamation: scan all hazard records; free retired objects not appearing in any hazard slot.

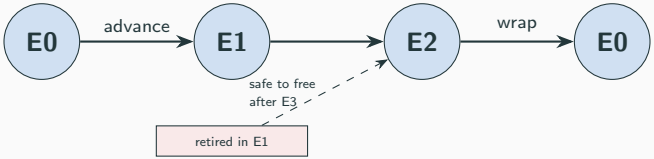


+ Deterministic, no epochs

– Scanning is $O(\text{threads} \times \text{retired})$

Epoch-Based Reclamation (EBR)

Idea *(Fraser 2004; Dice, Shalev & Shavit 2009)*: use a *global epoch* counter. Threads report the current epoch when they enter a critical section ("pinning").



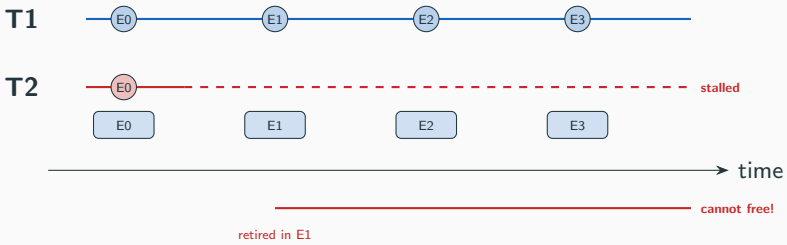
Rule: an object retired in epoch E can be freed once every active thread has announced epoch $\geq E + 2$.

\+ Very low overhead on fast path (read a counter)

− A single stalled thread blocks all reclamation

```
// pseudo-code
fn read(map, key) {
    guard = collector.pin(); //
        announce E
    let val = map.get(key);  // safe!
    drop(guard);             // unpin
}
```

EBR: The Stalled-Thread Problem



Stalled or slow threads prevent the epoch from advancing, causing **unbounded memory accumulation**. This is the key motivation for Hyaline.

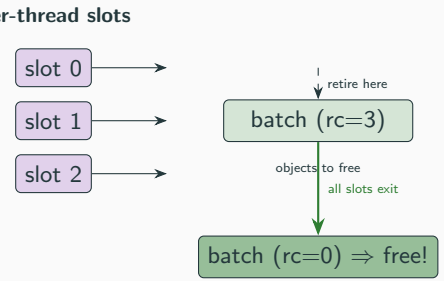
Hyaline: Batched Reference Counting

Idea *(Nikolaev and Ravindran 2021)*: no epochs; instead, use a *per-batch reference count* equal to the number of threads currently active.

Batch: all objects retired together in one operation share a single reference count — one atomic counter for the group, not one per object.

Key insight: the rc is set at retirement to the number of currently-active slots. Each slot decrements on exit. When rc hits zero, the batch is safe to reclaim.

+ No blocking from stalled threads: stalled threads simply don't decrement batches they never saw.



seize: Hyaline for Rust

seize is the memory-reclamation crate underlying *papaya*. It implements a variant of Hyaline.

Core API:

```
// One collector per data structure
let col = Collector::new()
            .batch_size(2000);

// Pin this thread
let guard = col.pin();

// Safe to read shared data
let val = map.get(key, &guard);

// Schedule node for reclamation
guard.defer_retire(ptr,
    seize::reclaim::boxed);
// freed when all guards from
// before this point are dropped
```

Design choices:

- Each map owns its `Collector`
- `LocalGuard`: thread-local cache (cheap re-pin)
- `OwnedGuard`: send across threads
- Batch size controls reclamation granularity (trading latency vs. throughput)
- No global lock on fast path

Limitation: the map *owns* the reclamation subsystem — you cannot share reclamation state across collections.

sdd: Epoch-Based Reclamation for SCC

The *SCC* (Scalable Concurrent Containers) crate uses *sdd* ("Safe Deferred Destruction"), a classical EBR implementation.

Three-epoch protocol:

- Epochs $0, 1, 2$ cycling
- Threads announce local epoch via `Guard`
- Retired objects carried in per-epoch bags
- Epoch advances when all threads agree
- Objects freed two epoch advances after retirement

Key types:

```
use sdd::{Owned, Guard,
          AtomicShared};

// Owned: heap allocation tracked
// by EBR (like Arc + deferred drop)
let node = Owned::new(value);

// Guard: pins current thread
let guard = Guard::new();

// defer_reclaim: schedule for
// freeing after epoch advancement
guard.defer_reclaim(ptr);
```

Technique Comparison

	HP	EBR	Hyaline	Tracing GC
Fast-path cost	scan	counter read	slot ref	none
Reclamation delay	scan	2 epochs	0 stalls	GC schedule
Stalled threads	ok	blocks all	ok	ok
Cycle collection	✓	✓	✓	✓
Memory overhead	$O(T)$	$O(T)$	$O(T)$	GC header
Programmer burden	high	medium	low	low

T = number of threads.

All three manual techniques require **discipline**: every path that drops a node must pass through the reclamation subsystem. Tracing GC removes that requirement entirely.

Concurrent Hash Maps in Rust

DashMap: Sharded Locking

Design: split the key space into N shards (default: $4 \times$ CPU count). Each shard is an independent `RwLock<HashMap<K,V>>`.

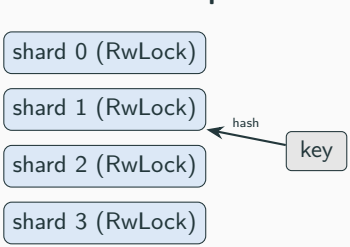
Reads: hash key $\rightarrow$ shard index $\rightarrow$ acquire R lock.
Writes: acquire W lock on that shard.

Strengths:

- Simple and correct by construction
- Leverages `parking_lot::RwLock` (faster than `std`)
- Fast for balanced workloads

Weaknesses:

- All writers to the same shard serialise
- Hot shards become bottlenecks
- No lock-free reads




```rust
let map = DashMap::new();
map.insert("key", 42);
// read lock on shard
let v = map.get("key");
```

Papaya: Lock-Free Reads with Hyaline

papaya *(Ahmed 2024)* provides a lock-free concurrent `HashMap` optimised for read-heavy workloads.

Key ideas:

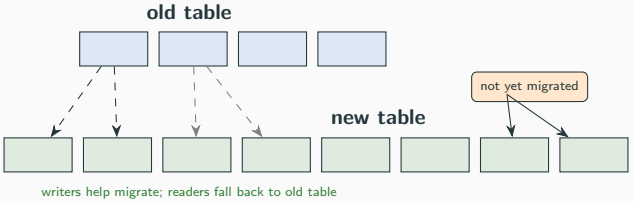
- Hash table stored as array of buckets, each an **atomic** linked list
- Reads are **lock-free**: CAS on bucket head only for writes
- Table **migration** on resize: incremental, non-blocking — readers help migrate entries
- Memory reclamation: *seize* (Hyaline variant)
- All node allocations go through `Box<T>`; retired via `defer_retire`

```
let map = papaya::HashMap::builder
    ()
  .collector(
    Collector::new()
      .batch_size(2000))
  .build();

// pin this thread to a seize guard
let guard = map.pin();
guard.get(&key);
guard.insert(key, val);
// guard drop => potential reclaim
```

Allocation: each map entry is a `Box<Entry<K,V>>` on the system heap. seize schedules `Box::from_raw` + drop when safe.

Papaya: Incremental Resizing



Migration proceeds *lazily*: each write operation migrates a bucket before performing its own work. This amortises resize cost without stopping readers.

PAX: Partition Attributes aXross

PAX is a *hybrid storage layout* for fixed-size records (Ailamaki et al., 2001).

Row layout: fields interleaved.

$[k_1 | v_1 | k_2 | v_2 | k_3 | v_3 | \ldots]$

Column layout: all keys, then all values.

$[k_1 | k_2 | \ldots] [v_1 | v_2 | \ldots]$

PAX: within each fixed-size *minipage*, keys packed first, values packed after.

$[k_1 | k_2 | k_3 | \ldots | v_1 | v_2 | v_3 | \ldots]$

Why PAX for hash buckets?

- Lookup probes *keys only* — more keys fit in a cache line
- Values fetched only on a *hit*, not during the probe scan
- Dense key array enables **SIMD** comparison (AVX2: 8–16 keys per instruction)
- Value layout stays cache-friendly for sequential scans

SCC HashMap: PAX Buckets + sdd

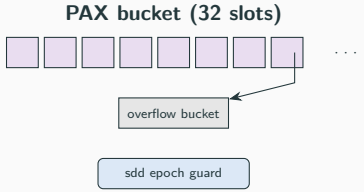
The *SCC* crate (*Scalable Concurrent Containers*) provides a richer family of lock-free structures: `HashMap`, `HashIndex`, `HashCache`, `TreeIndex`.

PAX storage format (per-bucket):

- Each bucket is a fixed-size array of 32 slots
- SIMD probing within a bucket
- Overflow via chained overflow buckets
- Bucket-level fine-grained locks (not map-level)

Reclamation via *sdd* (EBR):

- Old bucket arrays scheduled for deferred freeing
- `Owned<T>` tracks allocation; freed after epoch advance
- `Guard`: pins thread, provides `defer_reclaim`



Enable SIMD:

```
RUSTFLAGS='-C target-feature=+avx2'
```

SCC: Async and Sync Interfaces

SCC is unusual in offering both **blocking** and **async** (cooperative) interfaces for every operation.

```rust
use scc::HashMap;

let map = HashMap::default();

// synchronous
map.insert_sync("key", 1).ok();
map.read_sync("key", |k, v| *v);

// async (in Tokio etc.)
map.insert_async("key", 1)
   .await.ok();
map.read_async("key",
   |k, v| *v).await;
```

Why both?

- Blocked writers yield to the async runtime instead of spinning, preventing thread-pool starvation
- Same underlying data structure; the async wrappers are thin combinators over the bucket-level locks

HashIndex: read-optimised variant; *lock-free reads* using `AtomicRaw` + epoch-based reclamation.

Map Landscape Summary

	DashMap	**papaya**	**SCC**	**flurry**
Reads	RwLock shard	lock-free	bucket lock	EBR guard
Writes	RwLock shard	CAS	bucket lock	CAS
Resize	blocking	incremental	incremental	blocking
Reclamation	ownership (Drop)	seize (Hyaline)	sdd (EBR)	seize (Hyaline)
Async	no	no	yes	no
Rust idiom	ergonomic	guard-pinned	entry API	Java port

All of these use **manual reclamation discipline**: the implementer must ensure every removed node is passed through the reclamation subsystem. Forgetting means a use-after-free; premature freeing means the same.

Alloy: Tracing GC for Rust

Alloy: What and Why

Alloy is a fork of the Rust compiler that adds a first-class `Gc<T>` smart pointer backed by the Boehm–Demers–Weiser (BDW) conservative tracing GC.

Goals:

- Opt-in GC: existing Rust code unchanged
- `Gc<T>` is *Copy* — no clone overhead
- Cycle-safe: BDW traces all GC roots automatically
- Works with Rust's type system & borrow checker
- Finalizers run on a background thread

Non-goals

- Compacting GC (BDW is non-moving)
- Generational/incremental GC (BDW supports both; not yet explored)
- Precise (non-conservative) tracing
- Eliminating stop-the-world pauses (inherent to BDW)
- Replacing `Arc<T>` or `Box<T>`

BDW is **conservative**: it treats any word-aligned value that looks like a heap pointer as a root. No type information needed.

`Gc<T>`: The User-Facing API

```rust
#![feature(gc)]
use std::gc::Gc;

// Allocate in GC heap
let a = Gc::new(42_i64);
let b = a;  // Copy! no clone CAS

// Cycles are fine
struct Node {
    next: Option<Gc<Node>>,
}
let n = Gc::new(Node { next: None });
// BDW traces and collects cycles

// If T: Drop, finalizer is called
// on a background GC thread
```

Compared to `Arc<T>`:

	`Arc<T>`	`Gc<T>`
`Clone`	CAS	*no-op (Copy)*
`Drop`	CAS	*no-op (Copy)*
Cycles	hang	collected
Deterministic	yes	no
Send	✓	✓
Sync	✓	✓

The `Copy` property is the key win: no atomic operations on the hot path for concurrent data structures.

GcAllocator: Routing All Allocations to BDW

Alloy exposes GcAllocator as a GlobalAlloc and Allocator implementation:

```
#[global_allocator]
static A: GcAllocator = GcAllocator;
```

When GcAllocator is the global allocator, *every* allocation (Box, Vec, Arc, ...) goes through BDW.

Internally:

```
unsafe fn alloc(&self, layout: Layout)
  -> *mut u8
{
    // fast path: low alignment
    GC_malloc(layout.size())
    // high alignment: posix_memalign
}
```

Finalizer registration (inside Gc::new):

```
if !needs_finalizer::<T>() {
    // no Drop: skip registration
    return Box::leak(
        Box::new_in(GcBox { value },
                    GcAllocator)).
                        into();
}
// monomorphised drop glue
unsafe extern "C"
fn finalizer_shim<T>(obj: *mut u8,_)
{
    drop_in_place::<GcBox<T>>(
        obj as *mut GcBox<T>);
}
GC_register_finalizer_no_order(
    ptr, Some(finalizer_shim::<T>),
    ...);
```

Finalizer Safety Analysis (FSA)

Problem: finalizers run on a *background thread*. If a finalizer touches a `Gc<T>` field, that field may already have been finalized — use-after-free!

Solution: Alloy introduces the `FinalizerSafe` marker. `Gc<T>` is *not* `FinalizerSafe`:

```
impl<T: ?Sized>
    !FinalizerSafe for Gc<T> {}
```

`Gc::new` is annotated `#[rustc_fsa_entry_point]`. The compiler checks that nothing reachable from the drop chain of `T` dereferences a `Gc<T>`.

Escape hatches:

- `NonFinalizable<T>`: skip finalization entirely
- `FinalizeUnchecked<T>`: opt out of FSA (unsafe)

```
// compile error: Gc<T> in Drop
struct Bad {
    inner: Gc<String>,
}
impl Drop for Bad {
    fn drop(&mut self) {
        // ERROR: *self.inner
        // may already be finalized
        println!("{}", *self.inner)
            ;
    }
}
```

FSA is a **compile-time** guarantee — no runtime overhead.

BDW: Conservative Tracing

BDW operation:

1. **Roots**: scan stacks, registers, globals for anything that looks like a GC heap pointer
2. **Mark**: trace transitively from roots
3. **Sweep**: free unmarked objects; call registered finalizers
4. **Finalizers**: run on a dedicated background thread in non-deterministic order

"Conservative": no type information; any word that could be a GC pointer is treated as one. Occasional false retention, but no false frees.

Configuration in Alloy:

```
pub fn init() {
    // single mark thread
    // (avoids parallel-mark
    //  lock contention)
    GC_set_markers_count(1);
    GC_init();
}

pub fn force_gc() {
    GC_gcollect();
}
```

Alloy configures BDW for always-multithreaded mode (all threads registered automatically via `GC_pthread_*` wrappers).

Premature Finalization Prevention

Compilers may eliminate a reference earlier than its source-level lifetime to free a register. With GC, this can allow the object to be collected and finalized too soon.

Solution: when `premature-finalizer-prevention` is enabled, `Gc<T>` implements `Drop` with a compiler barrier:

```
impl<T: ?Sized> Drop for Gc<T> {
    fn drop(&mut self) {
        // compiler barrier: keep
        // this reference alive until
        // the source-level drop point
        GC_keep_alive(self);
    }
}
```

Special case

`Gc<T>` is `Copy`, but Rust normally forbids implementing both `Copy` and `Drop`.

Alloy adds *compiler support* to allow this special case: the drop is a barrier, not a user-observable destructor.

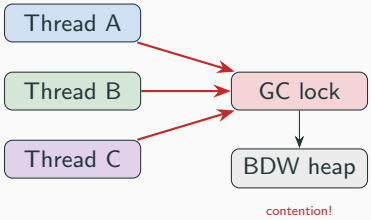
This is one of several places where Alloy requires **compiler modifications**, not just library changes.

Our Approach: Hybrid Allocation

The Bottleneck: BDW Allocation Under Concurrency

Observation: routing *all* allocations through `GcAllocator` (by setting it as the global allocator) creates bottlenecks:

- BDW uses **thread-local allocation** blocks, but still requires a **global lock** to acquire new blocks
- The **finalization queue** is shared; high-finalization workloads (large maps with frequent removes) saturate it
- Data-structure *infrastructure* (*i.e.*, bucket arrays, resize metadata) does not need GC, yet pays GC overhead if all allocations are routed through BDW



Three Allocators: Upstream Alloy + Our Additions

Upstream Alloy (Hughes and Tratt 2025) provides `GcAllocator`. **We added** `RootAllocator` and `GcUntracedAllocator` to `std::gc`:

Allocator	BDW function	Scanned?	Collectable?	Source
`GcAllocator`	`GC_malloc`	yes	yes	upstream
`RootAllocator`	`GC_malloc_uncollectable`	yes	**no**	**ours**
`GcUntracedAllocator`	`GC_malloc_atomic`	**no**	yes	**ours**

`RootAllocator`: in the BDW heap, **uncollectable** — BDW scans it for `Gc<T>` pointers (keeping them live) but never frees it; explicit `gc_free` required.

Critical: infrastructure on the system heap (outside BDW) would *not* be scanned, causing false collection of values reachable only through it.

`GcUntracedAllocator`: `GC_malloc_atomic` — BDW skips scanning the block entirely. Used with `GcUntraced<T>` for pointer-free values.

Together with `GcAtomic<T>` (also ours), these give fine-grained control over scan/collect behaviour — the key to avoiding BDW overhead on map infrastructure.

The `Untraced` Auto Trait (our addition)

A new *unsafe auto trait* in `core::marker` — like `Send`/`Sync`, auto-derived from fields:

```
pub unsafe auto trait Untraced {}
```

The three GC pointer types opt *out*:

```
impl<T: ?Sized> !Untraced for Gc<T> {}
impl<T> !Untraced for GcAtomic<T> {}
impl<T> !Untraced for GcUntraced<T> {}
```

So `struct Foo { x: Gc<Bar> }` is **not** `Untraced` automatically.

Two levels of enforcement:

- `GcUntraced<T>` requires `T: Untraced` — **compile-time**, won't compile if `T` has traced fields
- `Box::new_in` checks `mem::traced_of::<T>()` vs `Allocator::is_tracing()` — **runtime** panic on mismatch

Open: types generic over `T` (traced or not depending on the type argument) have no compile-time solution yet — the allocator direction is the current exploration.

GcAtomic<T>: A GC-Aware Atomic Pointer (our addition)

Problem: AtomicPtr<T> bypasses Alloy's type system, FSA, and Untraced tracking.

Solution: GcAtomic<T> wraps an AtomicPtr as a first-class GC type:

```
pub struct GcAtomic<T> {
    ptr: AtomicPtr<GcAtomicBox<T>>
}
impl<T> !Untraced for GcAtomic<T> {}
impl<T> !FinalizerSafe for GcAtomic<T>{}

pub fn null() -> Self
pub fn new(value: T) -> Self
pub fn load(&self,ord:Ordering)->*mut T
pub fn store(&self,ptr:*mut T,ord:
    Ordering)
pub fn compare_exchange(...)-> Result
    <...>
```

Why it matters:

- Lives in the GC heap; BDW scans the AtomicPtr as a root — the pointed-to GcAtomicBox<T> is also collected
- Drop glue elided by a MIR pass when T has no finalizer (same optimisation as Gc<T>)
- Used in scc-alloy: replaces sdd::AtomicShared<BucketArray> for the current bucket array pointer

scc-alloy: `GcAtomic` Replaces `sdd` for Resize

In original SCC, the *bucket array* lifecycle is managed by *sdd*: defer old array free for two epochs.

Original SCC (sdd):

```
struct HashMap<K, V, ..> {
    // AtomicShared:
    // sdd reference-counted
    // atomic pointer
    bucket_array: AtomicShared<
        BucketArray>,
}
// on resize: retire old array to sdd
fn defer_reclaim(old, guard: &Guard) {
    guard.defer_drop(old); // sdd
        schedules
}
```

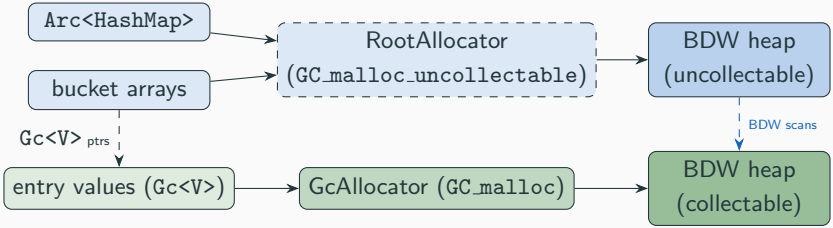
scc-alloy:

```
struct HashMap<K, V, ..> {
    // GcAtomic: GC'd + CAS-able
    bucket_array: GcAtomic<
        BucketArray>,
}
// on resize: just swing the pointer
// old array becomes unreachable
// BDW collects it at next GC
// no defer_reclaim needed!
```

Similarly, `BucketArray.linked_array` is a `GcAtomic<BucketArray>`: once migration completes and the link is cleared, BDW reclaims automatically.

The Hybrid Strategy: `RootAllocator`

Key insight: GC is needed for *values* stored in the map. The *map structure* (bucket arrays, outer `Arc`) must be in the BDW heap (so pointers are scanned) but must not be collectable.



The scan arrow is key: BDW traces through uncollectable objects to find `Gc<V>` pointers, keeping values live as long as the map exists.

papaya-alloy: Replacing seize with `Gc<T>`

Original papaya:

- Each entry: `Box<Entry<K,V>>` (system heap)
- Retired via `guard.defer_retire(...)`
- seize tracks which guards are active, frees batches

papaya-alloy:

- Each entry: `GcUntraced<Entry<K,V>>` (BDW atomic heap)
- *No* explicit retirement
- BDW traces the table, frees unreachable entries
- seize **removed entirely**

```
// papaya-alloy: entry allocation
use std::gc::GcUntraced;

// insert: BDW atomic heap
// (no interior GC pointers)
let entry =
  GcUntraced::new(Entry { key, value
      });

// CAS into table
table.store(slot, entry);

// remove: nothing explicit --
// BDW collects when unreachable
```

papaya-alloy: Before and After

What was removed:

- `seize` dependency
- Guard pinning and `defer_retire` call sites
- Collector configuration at map construction

What replaced it:

- `GcUntraced<Entry<K,V>>` for entries
- BDW conservative scan of the live table
- Nothing to configure or call on removal

Before (papaya + seize):

```
// insert: heap-allocate entry
Shared::boxed(Box::new(value));
// remove: schedule for freeing
guard.defer_retire(ptr,
    seize::reclaim::boxed);
```

After (papaya-alloy):

```
// insert: GC-allocate entry
Gc::new(value);
// remove: nothing -- BDW traces
// the table and collects entries
// when they become unreachable
```

papaya-alloy: Map Container Allocation

The map *container* itself (the `HashMap<K,V>` struct, its internal bucket arrays) uses `RootAllocator`:

```
pub struct PapayaAlloyTable<K, H>(
    Arc<papaya_alloy::HashMap<K, Value, H>, std::gc::RootAllocator>,
//                                          ^^^ GC_malloc_uncollectable
);
```

Constructed with `Arc::new_in(map, RootAllocator)`. The original papaya passes a `seize::Collector` here instead; papaya-alloy omits it entirely — no reclamation subsystem to configure.

scc-alloy: GC Replaces sdd for Bucket Arrays

SCC uses *sdd* to defer freeing of old bucket arrays during resize. scc-alloy replaces this with `GcAtomic<T>`.

scc-alloy approach:

- Entry values: `Gc<V>` (BDW heap, traced)
- Bucket array pointer: `GcAtomic<BucketArray>` (BDW heap, atomic CAS, BDW collects old arrays on resize)
- Migration chain (`linked_array`): also `GcAtomic<BucketArray>`
- Container (`Arc<HashMap>`): `RootAllocator` (uncollectable, scanned)
- sdd **removed** for all ported sync `SccAlloyMap` methods

```
// scc-alloy adapter
pub struct SccAlloyMap<K, H>(
    Arc<
      scc_alloy::HashMap<K,Value,H
          >,
      std::gc::RootAllocator,
    >,
);

fn with_capacity(cap: usize)
  -> Self
{
    Self(Arc::new_in(
        HashMap::
            with_capacity_and_hasher
            (
            cap, H::default()),
        std::gc::RootAllocator,
    ))
}
```

Design Summary: What Changed

Aspect	Original	Alloy version	Effect
Entry values	`Box<V>` (sys)	`Gc<V>` (`GC_malloc`)	automatic reclaim
Value reclamation	seize / sdd	BDW tracing	no retire calls
Bucket array ptr	`sdd::AtomicShared`	`GcAtomic<BucketArray>`	CAS + GC collect
Container alloc	`Arc<M>` (sys)	`Arc<M, RootAlloc>` (`GC_malloc_uncollectable`)	scanned, not collected
Retire call sites	many	none (papaya); partial (scc)	simpler code
Cycle handling	impossible	free	BDW handles cycles

The programmer no longer needs to ensure that every removal path calls `defer_retire`. BDW's conservative scan of the live table provides the liveness information automatically.

Remaining Challenges & Open Questions

What we know works:

- papaya-alloy: correct under the bustle benchmark harness
- scc-alloy: correct for sync operations
- Hybrid allocation eliminates BDW allocation hot-path contention
- FSA catches unsafe Drop implementations at compile time

Open questions:

- **GC pause latency**: BDW stop-the-world pauses vs. seize/sdd incremental reclamation
- **Memory overhead**: conservative false retention under high table density
- **Finalizer throughput**: BDW single finalizer thread vs. sdd deferred freeing
- **Precise GC**: can we annotate layout to help BDW avoid false retention?
- **Multi-map workloads**: BDW is global; seize is per-map. Interactions?

Benchmarking Methodology

We use the *bustle* harness (a port of libcuckoo's Universal Benchmark) via *conc-map-bench* (Wejdenstål et al.; we forked to add Alloy adapters). Maps prefilled to 50,000 entries; 10^9 operations per run.

Workloads (read% / put%):

- 90% read, 10% put
- 80% read, 20% put
- 70% read, 30% put
- 60% read, 40% put
- 50% insert, 50% remove

Machines:

- Intel Xeon Gold 5118 Skylake (12 cores, 2 threads/core, 1 NUMA node)
- AMD EPYC 9534 Zen 4 (64 cores, 2 threads/core)

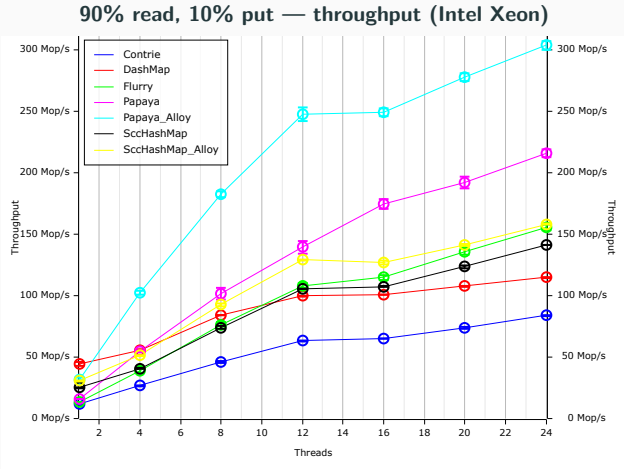
Maps compared:

- **DashMap**: sharded locking baseline
- **Flurry**, **Contrie**: seize / EBR baselines
- **Papaya**: seize (Hyaline), no GC
- **Papaya_Alloy**: `Gc<T>`, hybrid allocators
- **SccHashMap**: sdd (EBR), no GC
- **SccHashMap_Alloy**: `GcAtomic` bucket arrays
- **Papaya_Alloy_GcGlobal**: `GcAllocator` as global allocator

Preliminary Results (Intel Xeon)

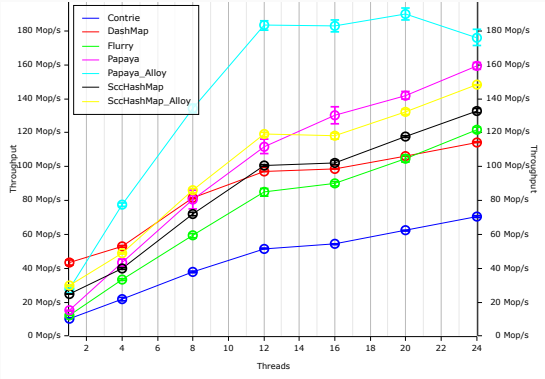
Throughput — 90% read, 10% put (Intel Xeon):

- Papaya_Alloy leads at low–mid thread counts; `defer_retire` bookkeeping overhead is gone
- SccHashMap_Alloy tracks SccHashMap closely
- Scaling begins to slow at higher thread counts — STW pauses visible but not yet decisive

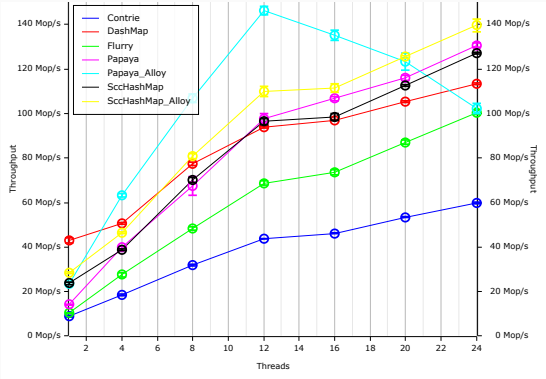


Intel Xeon: More Read-Heavy Workloads

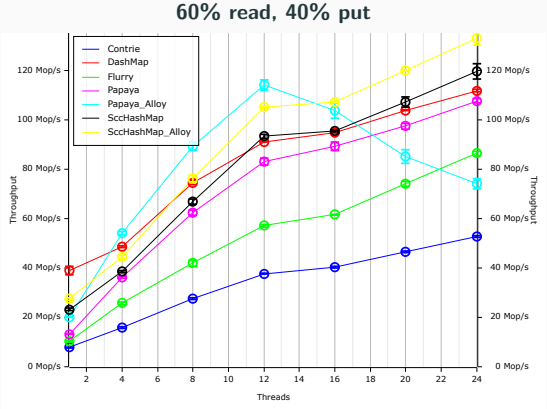
80% read, 20% put

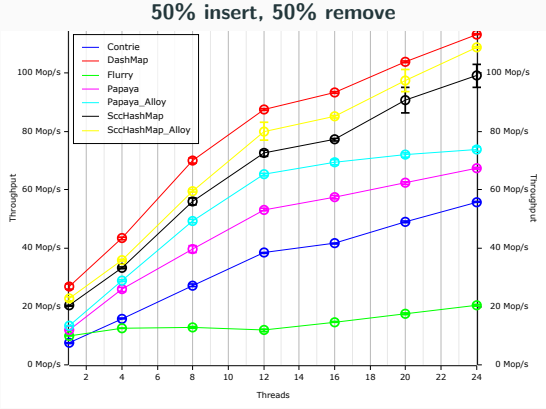


70% read, 30% put



Intel Xeon: Write-Heavy Workloads

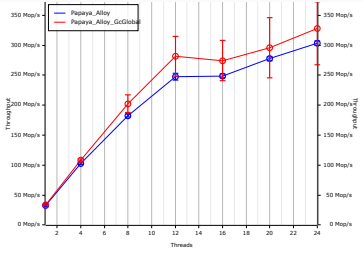




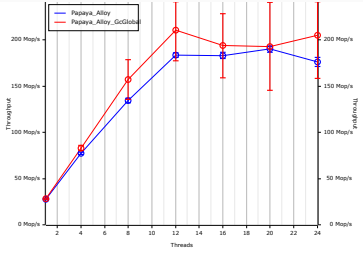
As write pressure increases, Alloy variants begin to fall behind — GC pauses more frequent; divergence most visible in the 50/50 workload.

Global GcAllocator vs Hybrid (Intel Xeon)

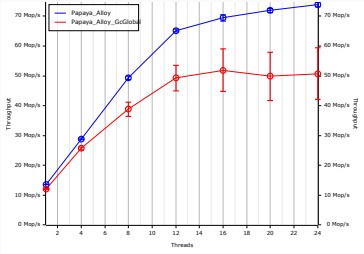
90/10 read/put



80/20 read/put



50/50 insert/remove



Papaya_Alloy (hybrid: 12 MB BDW + standard allocator for non-GC) vs **Papaya_Alloy_GcGlobal** (16 GB BDW heap). GcGlobal's large heap arises because setting `GcAllocator` as the global allocator diverts *all* standard Rust allocations through BDW, not just the GC'd entries. The hybrid avoids this entirely.

Heap Footprint (Papaya_Alloy, 90/10, Intel Xeon)

Threads	GCs	RSS (mim.)	RSS (dlm.)
1	394	8 022 MB	18 MB
4	408	8 022 MB	18 MB
8	408	16 024 MB	18 MB
12	428	17 358 MB	19 MB
16	428	16 026 MB	19 MB
20	428	16 027 MB	19 MB
24	415	17 361 MB	19 MB

BDW heap (hybrid): **12 MB** regardless of thread count.

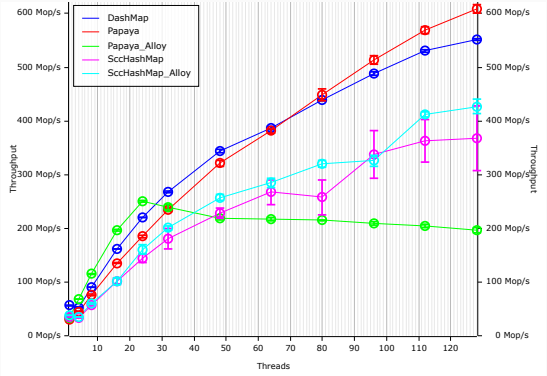
mimalloc retains OS pages aggressively: RSS climbs to 16+ GB — dominated by the non-GC allocator, not BDW itself.

dlmalloc returns pages promptly: RSS stays at ∼19 MB across all thread counts.

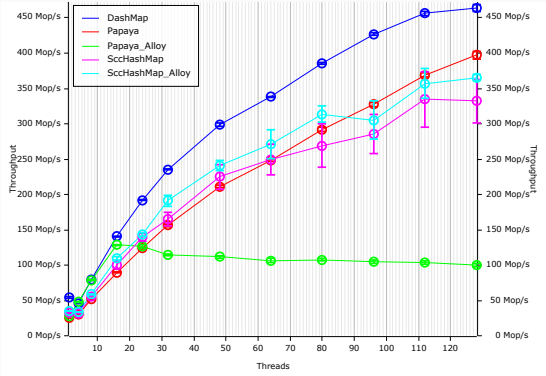
GC count (∼400 per iteration) is stable — the GC cadence does not worsen with more threads.

AMD EPYC Zen 4 (64 cores)

90% read, 10% put



80% read, 20% put



At 64 cores the gap is unambiguous: **Alloy variants plateau while DashMap and Papaya keep scaling**. Two compounding factors: **STW pauses** (BDW halts all threads) and **allocator contention** (BDW's global lock under many concurrent threads).

Addressing the Scaling Bottleneck

Root causes of the GC-induced plateau:

- BDW stop-the-world pauses halt *all* threads simultaneously — pause cost grows with heap size and thread count
- GcGlobal's 16 GB heap: `GcAllocator` as global allocator diverts all standard Rust allocations through BDW — not just GC'd entries — inflating the heap and worsening pauses

Near term:

- **Minimise BDW heap**: hybrid allocation + `traced` branch type enforcement keeps only true `Gc<T>` nodes in BDW
- **BDW incremental mode**: `GC_enable_incremental()` spreads collection work across mutations
- **Parallel marking**: BDW `--enable-parallel-mark` uses multiple

Longer term:

- **Safepoint integration**: pause threads only at quiescent points between map operations, not mid-CAS
- **Generational GC**: BDW supports it; short-lived entries collected cheaply in the nursery without a full-heap scan
- **Precise tracing**: layout annotations eliminate false retention and reduce scan

Summary

What we did:

- Described three manual reclamation techniques: hazard pointers, EBR, Hyaline
- Showed how papaya (seize/Hyaline) and SCC (sdd/EBR) implement these in Rust
- Described Alloy: `Gc<T>` via BDW in Rust, with FSA for finalizer safety
- Introduced the hybrid allocation strategy (`RootAllocator` + `GcAllocator`) to avoid BDW allocation bottlenecks
- Showed how both papaya and SCC were adapted with significantly simpler reclamation logic

Take-aways:

1. Manual reclamation is correct but complex: every removal path must call the right retire function
2. `Gc<T>` eliminates that discipline: liveness is determined by the GC, not programmer annotations
3. The key to avoiding BDW overhead: allocate *infrastructure* outside the GC heap
4. FSA gives compile-time safety for finalizers — no runtime overhead

Status: correctness confirmed; performance benchmarks in progress.

Thank you!

Questions?

antony.hosking@anu.edu.au

Backup: BDW FFI Surface

Allocation:

- `GC_malloc(size)` — traced (collectable, scanned)
- `GC_malloc_uncollectable(size)` — scanned, not freed by GC
- `GC_malloc_atomic(size)` — collectable, not scanned
- `GC_posix_memalign(&ptr, align, size)`
- `GC_realloc(ptr, size)`
- `GC_free(ptr)`

Finalization:

- `GC_register_finalizer_no_order(ptr, fn, ...)`
- `GC_finalized_total()` — stat

GC control:

- `GC_init()` — must call before any allocation
- `GC_gcollect()` — force a collection
- `GC_set_markers_count(n)` — parallel mark threads
- `GC_get_gc_no()` — collection count

Threads:

- `GC_pthread_create` / `GC_pthread_join`
- `GC_thread_is_registered()` — sanity check
- `GC_keep_alive(ptr)` — compiler barrier

Backup: GcStats — Profiling Alloy

```rust
#[cfg(feature = "log-stats")]
pub struct GcStats {
    pub finalizers_registered: u64,  // Gc::new() with Drop
    pub finalizers_completed:  u64,  // BDW GC_finalized_total()
    pub allocated_gc:          u64,  // GcAllocator::allocate() calls
    pub allocated_boxed:       u64,  // Box::new() calls
    pub allocated_arc:         u64,  // Arc::new() calls
    pub num_gcs:               u64,  // GC_get_gc_no()
}
```

Useful for diagnosing:

- **Finalizer backlog**: `registered` $\gg$ `completed` $\Rightarrow$ finalizer thread is behind
- **GC frequency**: `num_gcs` over time
- **Allocation split**: ratio `allocated_gc` vs. `allocated_boxed` tells you how much is going through BDW

Related Work

Memory reclamation:

- Epoch-based (EBR): Fraser 2004
- Hazard pointers: Michael 2004; ISO C++26 proposal
- Hyaline: Nikolaev and Ravindran 2021
- DEBRA+ (signal-based EBR): Brown 2015
- IBR (interval-based): Wen et al. 2018

GC in systems languages:

- BDW GC: Boehm and Weiser 1988
- Alloy (`Gc<T>` for Rust): Hughes and Tratt 2025
- Go runtime: tricolor concurrent mark-sweep
- D language: optional GC + manual

Concurrent data structures:

- Lock-free linked lists: Harris 2001
- Split-ordered hash tables: Shalev and Shavit 2006
- Java `ConcurrentHashMap`: Doug Lea, JDK
- flurry (Rust): Java CHM port using seize (Hyaline)

Benchmarks:

- conc-map-bench: Wejdenstål 2021

GC + concurrent structures:

- Managed runtimes (*e.g.,* JVM, .NET) use GC as the universal reclamation mechanism
- Our work: opt-in GC in a systems language; hybrid allocation to manage overhead

References i

- Boehm, Hans-Juergen and Mark Weiser (1988). **"Garbage Collection in an Uncooperative Environment".** In: *Software: Practice and Experience* 18.9, pp. 807–820. DOI: 10.1002/spe.4380180902.
- Brown, Trevor Alexander (2015). **"Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way".** In: *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 261–270. DOI: 10.1145/2767386.2767436.
- Fraser, Keir (2004). **Practical Lock-Freedom.** Tech. rep. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. DOI: 10.48456/tr-579.
- Harris, Timothy L. (2001). **"A Pragmatic Implementation of Non-Blocking Linked-Lists".** In: *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*. Vol. 2180. Lecture Notes in Computer Science. Springer, pp. 300–314. DOI: 10.1007/3-540-45414-4_21.
- Hughes, Jacob and Laurence Tratt (2025). **"Garbage Collection for Rust: The Finalizer Frontier".** In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2, pp. 3588–3614. DOI: 10.1145/3763179.

References ii


Michael, Maged M. (2004). **"Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects".** In: *IEEE Transactions on Parallel and Distributed Systems* 15.6, pp. 491–504. DOI: 10.1109/TPDS.2004.8.


Nikolaev, Ruslan and Binoy Ravindran (2021). **"Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures".** In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. DOI: 10.1145/3453483.3454090.

Shalev, Ori and Nir Shavit (2006). **"Split-Ordered Lists: Lock-Free Extensible Hash Tables".** In: *Journal of the ACM* 53.3, pp. 379–405. DOI: 10.1145/1147954.1147958.


Wejdenstål, Joel (2021). **conc-map-bench: Concurrent Hash Map Benchmarks for Rust.** https://github.com/xacrimon/conc-map-bench.

References iii

Wen, Haosen et al. (2018). **"Interval-Based Memory Reclamation".** In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/3178487.3178488.