Going Where No GC Has Gone Before

A No-Tradeoff Memory System for Bosque

Anthony Arnold
anthony.arnold@uky.edu
University of Kentucky
Lexington, Kentucky, USA

Mark Marron
mark.marron@uky.edu
University of Kentucky
Lexington, Kentucky, USA

Abstract

Garbage Collectors (GCs) are a critical component of a modern application stack. Long pauses, large memory consumption, and high CPU usage can unexpectedly occur with certain workloads or series of events. These behaviors can make systems unresponsive, make it impossible to run them in resource-constrained environments, and are often very difficult to debug and fix – as their appearance may be intermittent. In fact recent theoretical work has shown what, for existing mainstream languages, these issues are unavoidable and, regardless of the GC design or implementation, there will always be workloads that cause them to occur!

Intriguingly, the pathological and scenarios that are needed to cause these GC behavioral issues are a result of a specific set of language features – mutability and cyclic data structures. In this paper we present a novel language and garbage-collector co-design, The collector is designed leverage programming language features to construct a system with *provably* bounded collection pauses, incurs a fixed-constant memory overhead, and ensures starvation freedom for the application.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: Garbage Collection, Programming Languages, Runtime Systems, Bosque

ACM Reference Format:
Anthony Arnold and Mark Marron. 2026. Going Where No GC Has Gone Before: A No-Tradeoff Memory System for Bosque. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3814942.3816138

1 Introduction

A key-performance-indicator (KPI) for many applications is the 99th (or 95th) response percentile latency – that is, the time it takes for the application to respond to a user request 99% of the time. This is a critical metric as these *tail-latency* events are often pain points for users and, once encountered, lead to disengagement [23]. Unfortunately, these tail-latency events are often intermittent, involve multiple events, and even different subsystems [8, 22]. These features combine to make them very difficult to diagnose and resolve.

Some sources of tail-latency are irreducible parts of application execution, such as connection latency, shared resource contention, or hardware failures. However, latency from these sources is often amplified by runtime characteristics. For example, a network stall that leads to requests backing up, which leads to many objects being promoted into old GC generations, leading to a long GC pauses during whole heap collection, causing more requests to back up, and so on. The triggering event for the latency spike is an intermittent network stall but the amplification and eventual failure is in the GC behavior.

These behaviors can make systems unresponsive, make it impossible to run them in resource-constrained environments, and as their appearance may be intermittent, are often very difficult to debug. Recent work [29] has theoretically demonstrated that it is impossible for (mainstream) languages with imperative features to simultaneously ensure bounded pause times and starvation-freedom without incurring large performance penalties (thrashing) in other areas. However, Bosque [18], which represents a new viewpoint for programming languages, provides a unique opportunity to rethink the design of the memory management system.

This paper introduces a new garbage collector Pando which is the first language/runtime/gc combination capable of satisfying the *no-tradeoff memory subsystem happiness* property (Theorem 5). Specifically, this work takes a well known GC design, a copying collector for young objects and a reference counting collector for old objects [2, 5], and simplifies the implementation using the distinctive aspects of the Bosque programming language to construct a specialized garbage collector with the following novel properties:

- **Bounded Collector Pauses:** The collector only requires the application to pause for a (small) bounded period that is proportional to the size of the nursery.
- **Starvation Freedom:** The collector can never be outrun by the application allocation rate and will always satisfy allocation requests (until true exhaustion).

ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816138

- **Application Code Independence:** The application code does not pay any cost, *e.g.* read/write barriers, remembered sets, *etc.*, for the GC implementation.
- **Constant Memory Overhead:** The reserve memory required by the allocator/collector is bounded by a (small) constant overhead.

We show that such a system is possible and present a practical design & implementation of the Pando collector in Sections 3 and 4. Empirically, we validate (Section 6) that this design is effective with 50th percentile GC pause times less than 133 µs, a 99th percentile pause time of under 300 µs and, a reserve memory overhead proportional to 8 MB. In addition, the GC design ensures that each collection can always recover (if possible) sufficient memory to cover a full cycle of allocation requests (Section 3). Finally, as shown in Sections 3 and 4, application (mutator) code does not need to be modified with GC support operations (*e.g.* write barriers) and the algorithm works with conservative collection [28].

In summary the key contributions of this paper are:

- The formalization of the *no-tradeoff memory subsystem happiness* property, along with associated theorems and proofs, as a key objective for modern language/runtime/gc systems (Section 5).
- Demonstration, via a novel GC construction (Sections 3 and 4), that existing impossibility results [26, 29] do not apply to languages with the features of Bosque.
- An experimental evaluation of the collector showing that, in addition to its theoretical guarantees, the combined language/runtime system achieves (very) low and predictable pause times along with low memory overheads in practice (Section 6).

2 Bosque Background

The goal of the Bosque project is to create a programming system that is optimized for reasoning – by humans, symbolic analysis tooling, and AI agents (Large Language Models in particular) [18, 19]. The approach taken by Bosque is to identify and remove features or concepts that complicate reasoning and are frequent causes of software faults. Although the initial motivation of this work was focused on software assurance and quality, these same simplifications also provide strong guarantees about how memory can be allocated, organized, and used at runtime as well.

At the core of Bosque is a let-based functional language with a nominal type system for declaring datatypes. A sample Bosque program for computing the largest low-high temperature range in a list is shown in Figure 1.

The first declaration in the code in Figure 1 is an `entity` declaration of a composite datatype `TempRange` that has two fields: `low` and `high`, both of type `Int`. The function `maxRange` takes a list of `TempRange` values and returns the one with the largest difference between the `high` and `low`

```
entity TempRange {
  field low: Int;
  field high: Int;
}

function maxRange(temps: List<TempRange>): TempRange {
  return temps.maxElement(pred(t1, t2) => {
    return t1.high - t1.low < t2.high - t2.low
  });
}

maxRange(List<TempRange>{
  TempRange{32, 50}, TempRange{40, 60}, TempRange{20, 30}
});

%% Result is TempRange{40, 60}
```

Figure 1. Max Temperature Range in the Bosque Programming Language.

temperature fields. The code uses a higher-order function `maxElement` that takes a predicate to compare `TempRange` values. The last expression in the code is a call to `maxRange` with a literal list of three `TempRange` values. The result of this call is a `TempRange` with the value `TempRange{40, 60}`. There are a number of aspects of this example that are interesting from a memory management perspective.

Immutability: The `entity` declaration in Bosque creates a new composite datatype and these are always immutable. A key implication of this fact is that once an entity value is created then fields (thus pointer targets) will never change.

Referential Transparency: The semantics of Bosque ensure that reference identity of values is never observable – either directly via equality tests or indirectly via mutation. Thus, the allocator has wide latitude in value placement – stack, heap, or inline allocated. Object relocation is guaranteed to be semantically safe.

Cycle Freedom: In combination with the immutability of entities, and careful definition of constructor semantics, the Bosque language ensures that all data structures are acyclic. This invariant allows us to utilize reference-counting techniques without concern for backup cycle-collection or other special case logic.

Non-Escaping Lambdas: Although Bosque supports first-class functions and higher-order functions, and applications use them heavily, the language semantics require them to be in direct argument position as literals or passed parameters. As a result a lambda function cannot escape from the scope in which it is defined and the compiler can fully monomorphize higher-order code.

Ropes and RRB-Vectors: Lists and Strings in Bosque are implemented via efficient tree-structures [31]. This design has the benefit that even large strings (or lists) are implemented as a tree of fixed-size chunks. Along with a closed-world compilation model, this implies that all allocation sizes can be computed at compile time and are all small values.

```
template <size_t K>
class Allocator
{
  FreeEntry<K>* freelist;

  ...

  void* alloc(Type* t)
  {
    entry = this->freelist;
    if(this->freelist == nullptr) {
      //collector is run from here if/as needed
      return allocSlow(type);
    }

    this->freelist = this->freelist->next;
    return INIT_META(entry, type);
  }
};

...

void collect()
{
  markRoots();
  markHeap();
  processMarkedYoung();

  computeDeadRootsForDecrement();
  processDecrements();
}
```

Figure 2. Top-level allocation (size-segmented) and collection algorithms. The `Allocator` class is templated on object sizes `K` and maintains a free-list of available locations, per memory page, to allocate in the `freelist`. The `collect` function shows the high-level steps of the collection algorithm which is run when the nursery is full (default 8 MB).

3 GC Algorithm Overview

The overall design of the garbage collector is based on a hybrid-generational approach [2, 5], with a copying nursery for young objects and promotion to a reference counted old-space. For the stack we use a conservative scan [28] while objects are handled precisely. The top-level algorithm for the allocator and collect phase are shown in Figure 2.

The code in Figure 2 shows the high-level flow of the allocator code. The closed world semantics of Bosque enable us to pre-compute the sizes of every allocation needed in an application and statically create dedicated (thread-local) allocators for each size class. To perform an allocation each `Allocator` maintains a page of memory to allocate from and this page is organized using a single free-list layout of available locations. When an allocation is requested, the allocator has a fast path of taking the head of the `freelist` and advancing the `next` pointer. If the `freelist` is empty (`nullptr`) then the allocator calls a slow path to allocate a new page of memory, or run a collection cycle, and initialize the freelist for that page. This design provides good baseline allocator behaviors in terms of inlineable fast-path allocation performance and, the use of size-segmented and per-page free-lists, provides good spatial locality for object allocations [6, 16].

In our collector design heap values can be in one of two logical regions, the nursery or the reference-counted old space. This is shown in Figure 3. In this design the roots can point to values in the nursery, the RC-heap, or other stack allocated values[1]. However, in contrast to most generational collectors, there may be pointers from the nursery to the RC-heap *but not* pointers from the RC-heap to the nursery. This hierarchical relation allows for simplified garbage collection, as objects in the nursery can be collected without concern for references from older objects which cannot exist!

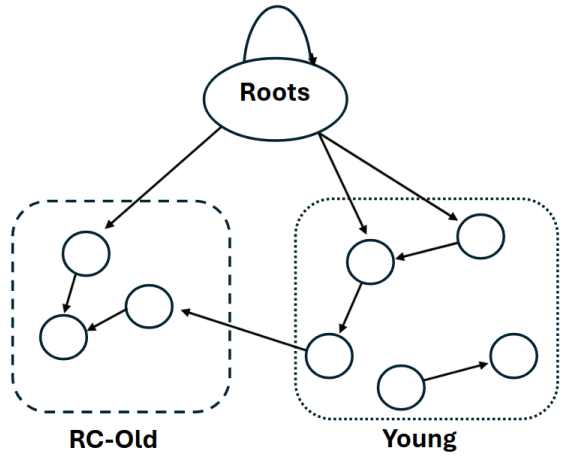
The code in Figure 2 shows the high-level overview of the collection algorithm. The first step is to mark all root references, which includes conservatively scanning the stack, global variables, and registers for pointers to heap objects. This code uses standard conservative scanning methods [21, 28]. The next step is to mark all reachable objects in the nursery, the `markHeap` call, which starts from the root set, traversing the object graph, and marking all reachable *young* objects (see Section 4 for details).

The state of (logical) memory model after these marking steps is shown in Figure 3b. In this figure the nursery is shown with a set of marked objects (shaded in black) that have been identified as reachable from the root set. Note that objects in the RC-old space are not marked and are never visited during the marking phase.

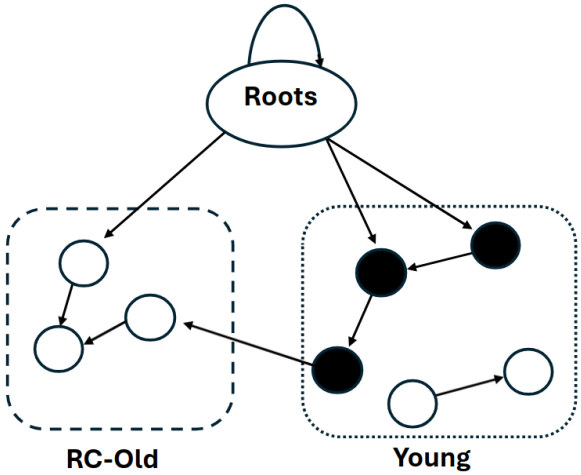
The next step is to process the marked young objects. For each marked object there are two possibilities, the first is that the object is referenced by a conservative root location, and the second is that the object is referenced only by other young objects. In the first case we cannot relocate the object, as the root location is conservative and cannot be updated. Thus, these objects are converted in place to a reference counted representation, shown as now black in Figure 3c. In the second case we can evacuate the object values to a compacting page, performing pointer forwarding as necessary, and converting them to reference counted representations (the objects in the dashed box). As each object is promoted to the RC-old space all fields are (precisely) scanned and reference counts for children are incremented (shown with black plus in Figure 3c).

Next the algorithm sweeps the now mostly (or entirely) evacuated nursery and rebuilds the free-lists for the next round of allocation – allowing the collector to re-use both completely empty and semi filled pages. As RC objects are not moved the ability to re-use partially filled pages is critical to avoid fragmentation. The sweep reclaims the unmarked dead objects and the space cleared by the evacuation. In our example, there is only one object that remains on the page,

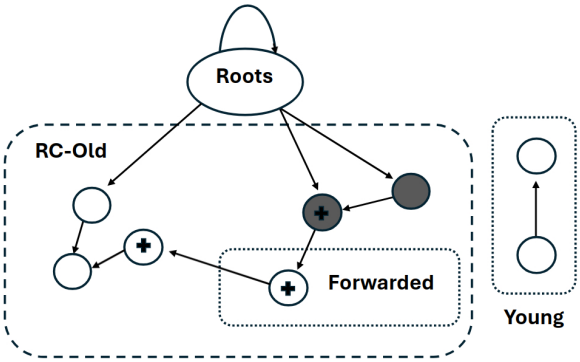
[1] As described in Section 4 roots may also point to the interior of stack values or heap allocated objects.



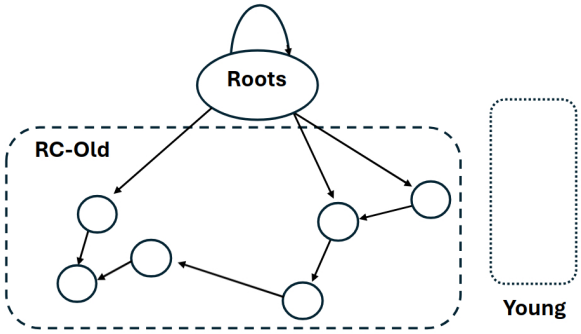
(a) State of memory at the start of a collection cycle. Note root references may point to stack locations, nursery, or RC-heap objects *but* pointers from the old RC-heap to the nursery are *not* possible.



(b) State of memory after marking young objects. The marked objects are shown colored in black.



(c) State of memory after processing marked objects. The young objects are evacuated and compacted to the old-space if possible (the forwarded region) or promoted to RC representations in place if they must be conservatively handled (*i.e.* there is a root reference to them). Objects with reference count increments are shown with a ⊕ symbol.



(d) State of memory after decrements and reset nursery. All of the unreachable objects from the young space have been reclaimed (via freelist management) and unreachable RC object with counts of 0 are collected.

Figure 3. Evolution of the (logical) memory layout during a collection cycle – from initial state, to marking young objects, evacuating young objects, adjusting reference counts, and finally resetting the nursery.

the root referenced object, while all other slots are evacuated or reclaimed (Section 4).

The final step in the collection algorithm is to process deferred reference count decrements and root reference status. Our implementation compares the root set from the previous collection with the root set identified in the current collection to determine which root references are no longer live. For each of these references the collector checks if the reference count has dropped to zero, in which case the object can be reclaimed. This reclamation is then a standard release walk of the object graph, decrementing reference counts and reclaiming objects as necessary. The state of the memory after this step is shown in Figure 3d. As each object is reclaimed, its memory is returned to the free list, in the appropriate page for later use.

The most notable feature of this algorithm is not what it has but what doesn't. Despite the use of a generational collector and reference-counting, there are no remembered sets, no write barriers, no backup cycle-collector, and no need for runtime support in the application code.

4 Pando Implementation

The logical model as described in Section 3 provides a clean separation between the nursery and RC-old space. However, a practical implementation must map these logical regions onto physical pages in memory in a manner that is efficient

for both allocation and collection. This section details that mapping, the management of physical pages, and key implementation details of the processing algorithms.

4.1 Memory Organization

The memory management system uses a set of thread-local page pools which feed a set of size-segregated allocators as shown in Figure 4. As Bosque provides a closed-world compilation model, and all lists/strings are implemented as trees/ropes [21, 31], the compiler can pre-compute all allocation sizes. Thus, we do not need to handle large, or variable sized, objects.

Each allocator is responsible for a given size class and, in the active allocation page (`allocPage` in Figure 4), the allocator maintains a null terminated free-list of available slots for allocation. The allocator also tracks pages used for evacuation during collection (`evacPage` in Figure 4) and a set of partially filled pages organized by utilization.

This free-list organization is required as the collector performs a conservative stack scan and the reference counted old-space objects are not movable. Since the collector may need to repurpose partially filled pages as allocation pages, we employ a free-list design where each entry is a fixed-size determined by its allocator. This ensures every free slot remains available for allocation, providing strong defense against fragmentation. The current implementation stores allocation metadata inline in the object header.

4.2 Allocation

The head of the active allocation free-list is stored in the thread-local static section. The combination of thread-locality, size-segregation, direct free-list access, and compile time known sizes allows us to emit an efficient allocator fast-path.

The fast path quickly checks if the free-list is non-empty, and if so takes the head of the free-list, advances the free-list pointer, and initializes the object metadata. If the free-list is empty then the allocator calls a slow-path function that will either scavenge a new page of memory from the page pool, or trigger a collection cycle to reclaim memory. As shown in Figure 4 the allocator maintains a list of partially filled pages in the page pool which can be selected as the new allocation page, falling back to requesting a new page from the OS if none are available.

In the Pando design we select from the set of partially filled pages the one with the lowest utilization for allocation and evacuation. This design ensures that pages are filled up as much as possible before new pages are requested from the OS, improving spatial locality and reducing overall memory usage. As reference-counted objects are naturally reclaimed over time the allocator will slowly compact pages and free up fully empty pages for return to the OS.

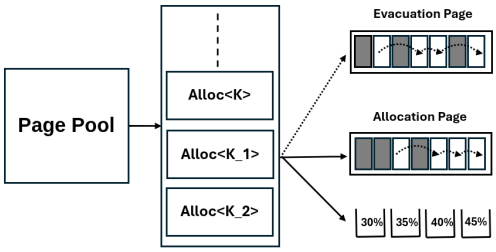


Figure 4. Pando Memory organization of pages, size-segregated allocators, and per-page free-lists. Each allocator manages a set of pages for allocation/evacuation, and partially filled pages for the RC-old space.

4.3 Young Collection

A collection trigger is included on the allocator slow-path and is controlled by a fixed nursery threshold, by default 8 MB. The stack scan is performed conservatively, in contrast to the precise object scan, and allows for stack allocated objects as well as interior pointers into stack/heap allocated objects. This design allows the compiler to avoid root-maps, allows aggressive value conversion (which is possible as Bosque is fully referentially transparent), and allows for easy inter-operation with core runtime C/C++ code.

Once the root set is computed the collector runs a tracing collection of the nursery. As the stack scan is conservative we cannot automatically evacuate all young objects, as we cannot safely relocate possible stack pointers. Thus, we either promote each young object in place, setting metadata to *old* with a initializing the reference count, or evacuate the object including setting the metadata and setting up a forwarding pointer. As we move objects we leverage the invariants that there are *no old→young* pointers and that the object graph is acyclic, which enables us to process young objects in reverse topological order and ensure all pointers are updated correctly. In the case of a *young→old* pointer we simply perform a reference count increment as the young object is currently being promoted to the RC old-space.

4.4 Reference Count Management

To start the RC phase, Pando begins by first comparing the root set collected in the previous collection with the set observed in the current collection. These sets are ordered based on address and a linear two-pointer walk is performed to determine whether a root object is present in the old set, but not the current set. If an object reference count is zero, and it is not referenced by a root, it is then inserted on a worklist for pending RC decrement and reclamation.

We process this worklist of now dead objects by decrementing the reference count of each child, enqueuing any dead child objects onto the same worklist, and then releasing the object itself. This worklist design allows us to control the amount of work done in a given collection, stopping if there

is a risk of a large deletion cascade. Currently, the Pando collector will stop processing after releasing $1.5 \times N$ objects, where N is the approximate number of objects distributed out of pages in the previous round of allocations. This partial processing with bounded work per cycle (or heartbeat [16]) ensures that the collector always blocks for a bounded period, while ensuring that reclaimed memory is proportional to the allocation rate, and ensures we never starve the application.

As the overall Pando collector is non-moving, and RC old-space objects from any page can die in any collection, then as time progresses pages will slowly drop in their utilization. As the allocator distributes objects it will require new pages to draw from. Thus, as shown in Figure 4, this set of partially filled pages is where the allocator can select a new allocation page from. For best performance we want to select a page that has a reasonable number of free slots, amortizing the cost of the slow-path work over as many allocations as possible.

To keep an accurate and up-to-date view of page utilization we update this information with defered live count updates during the RC processing. We organize our pages into utilization bins (5% increments) to prevent frequent moves between categories. Each bin is maintained as an approximate set based on page utilization. When a page's computed utilization crosses a predetermined threshold it is moved into the appropriate bin.

A key observation in this algorithm is that there is no cycle-collector needed as Bosque ensures that reference cycles cannot be created. This ensures that the RC processing is always over DAG and all decrements will eventually terminate with all unreachable object reference counts going to 0 and being reclaimed.

5 Theoretical Analysis

This section provides a theoretical analysis of the properties and guarantees of the overall memory management system.

Theorem 1 (Fixed Work Per Allocation)**.** *The work done by the garbage collector (GC) for each allocation is fixed and bounded by the function* $O(Field_{ct} * (Cost(Mark + Fwd + Inc + Dec) + Cost(Alloc + Copy + Release)))$*, and does not depend on the lifetime or behavior of the application.*

Theorem 1 states that the work done for each allocation is constant and is independent of the actions of the application code. The Alloc term represents the initial cost to allocate the object in the nursery and, is a simple pointer operation. After allocation the next lifecycle phase is collection and (possible) promotion to the old reference-counted generation. During a collection each allocated object (if still alive) will be visited during the marking and evacuation. During the marking phase we visit each field of the object, for $\text{Field}_{\text{ct}} * \text{Cost}(\text{Mark})$ work and during the evacuation we perform forwarding and (as needed) increments to field values pointers for $\text{Field}_{\text{ct}} * \text{Cost}(\text{Fwd} + \text{Inc}) + \text{Cost}(\text{Copy})$ work.

Once in the old-space cycle-freedom eliminates the need for a cycle-collector and immutability eliminates the need for a remembered set and the possibility of re-processing. These two features ensure that old-value objects will never be re-visited during liveness/reachability processing.

While the object is live in the old-space there may be increments and decrements of its reference count. The Inc operations are already accounted for as part of the promotion cost of predecessor objects. Decrements will only occur when a predecessor object is reclaimed. As with the Inc operations, the Dec operations can be accounted for as part of the predecessor reclaim cost. This cost is then $\text{Cost}(\text{Release}) + \text{Field}_{\text{ct}} * \text{Cost}(\text{Dec})$.

Theorem 2 (Bounded Collector Pauses)**.** *The garbage collector (GC) pause times are bounded by a constant factor K, determined by the size of the nursery, and are independent of application behavior or allocation rate.*

Theorem 2 states that the garbage collector (GC) pause times are bounded by a constant factor K determined by the size of the nursery. As the GC runs in two phases – processing, and possibly promoting, young objects in the nursery followed by checking root sets and updating reference counts – the cost of a collection is proportional to the work to process the nursery (K_{nursery}) and the cost to process roots and perform the reference operations (K_{old}).

The cost of the nursery component K_{nursery} is the cost to mark and evacuate live objects + the cost to sweep the nursery and rebuild free-lists. As the number of references/objects to process is fixed by the size of the stack/statics, as all objects are immutable and there is no remembered set, the number of objects marked and/or evacuated is bounded by the size of the nursery. The cost to sweep the nursery and rebuild free-lists is also, by construction, proportional to the size of the nursery.

The cost for performing root set reference operations, K_{old}, is a function of the number objects that had a root reference in the previous collection but do *not* have a root reference in the current stack. This is bounded by the size of the application stack (8 MB is standard default on linux). A naive decrement/release walk of these objects could, in the worst case, touch all objects in the heap. However, as described in Section 4, we use a classic control system to perform this work (possibly) over multiple collection cycles while ensuring that the overall cost remains bounded per collection cycle and, also, that we monotonically decrease the number of pending decrements.

Theorem 3 (Effective Collections)**.** *After a collection completes it has either reclaimed all unreachable objects or at least* $1.X\times$ *the size of the nursery – as a result the application allocation rate can never outrun the collector.*

Theorem 3 states that our collection strategy is always effective in reclaiming memory and that the application will

never starve for memory. As a corollary of the proof for Theorem 2 we have that either all reclaimable objects have been identified and recycled at the end of the collection *or* we have recycled at least the amount of memory allocated in the nursery plus a fraction given by X. This ensures that the application can never outrun the collector.

Theorem 4 (Fixed Memory Overhead *w.r.t.* Application Memory Usage)**.** *The memory overhead of the system* w.r.t. *to the max-live memory usage is given by a constant factor K determined by and proportional to the size of the nursery.*

Theorem 4 states that the memory overhead of the system *w.r.t.* to the max-live memory usage is given by a constant factor K determined by and proportional to the size of the nursery. As described in Section 3 the collector uses a nursery for the young space and all promoted objects are handled via a reference counting mechanism. Thus, the size of the old space is proportional to the size of dynamically live application objects. As the nursery is a fixed size, and the book-keeping data structures described in Section 4 are also proportional to the size of the nursery, the overall overhead is a constant factor *w.r.t.* the live application memory usage.

Theorem 5 (Memory Subsystem Happiness)**.** *The allocation rate of the application can never outrun the garbage collector (GC) and the collector only touches objects on the* fringe *of the old reference-counted space – thus starvation is eliminated, pause times are bounded, collection is always effective, and GC driven cache/page eviction is minimal.*

As a result of the above theorems we have Theorem 5. This summarizes the properties of the allocator/collector as a whole and the unique *no-tradeoff* nature of the result. Additionally, as the collector never touches objects in the old reference-counted space unless an object that is being promoted from the nursery references it or an object in the reference-counted space that references it is being reclaimed. Thus, these *fringe* objects are the boundary for memory touched by the collector in the old space – and any interior objects will never be accessed by the collector. This ensures that the collector does not thrash the memory subsystem.

Theorem 5 along with the analysis in [29] demonstrate that Bosque (and the Pando runtime) are unique in satisfying the *no-tradeoff memory subsystem happiness* property. Further, as proved in [29] it is theoretically impossible for any (mainstream) language with imperative features to achieve this level of performance without significant trade-offs!

6 Experimental Evaluation

As described in Section 1 our focus is on creating a software stack with highly-predictable performance characteristics. Thus, our evaluation in this section is split into three parts. The first section provides a general analysis of application performance and GC specific components. The second is a brief analysis of the sensitivity of the Pando collector to its configuration parameters. Finally, the third section provides a detailed set of experiments to evaluate end-to-end latency of the workloads as well the path-independence behaviors of the overall system.

As Bosque is a new language, there are limited existing applications to use as benchmarks. Thus, we focus on a well-known applications that have been re-implemented in Bosque. The first is the n-body simulations programs from the Computer Language Benchmarks Game [3]. The next is a raytracing program published on Microsoft's MSDN blog [12]. The db program is a Bosque implementation of the DB benchmark from SpecJVM 98 [30]. The final benchmark is the optimizer pass of the Bosque compiler (written in Bosque) which is the most complex of the benchmarks and largest Bosque program currently in existence.

The Pando collector is implemented in C++, 2.4kloc at present, and all evaluations use the standard release build configuration. All experiments were run on a system with an AMD Ryzen 9 9950X and 64GB of memory. The system is otherwise unloaded to minimize the impact of other workloads on the performance measurements. All runs use a default nursery size of 8 MB and a default page size of 8 kB.

Table 1 shows a set of static and dynamic statistics for each of the benchmarks. The code size column indicates the number of lines of Bosque source code for the benchmark while the types column indicates the number of distinct Bosque types created in the program. The next three columns provide dynamic memory statistics for the execution of the benchmark – specifically the total number of allocations and the total number of bytes allocated during the execution.

The statistics in Table 1 show that the benchmarks allocate heavily during execution, primarily a result of the direct implementation of immutable data structures in Bosque. In this design, each update performs a logarithmic number of copy allocations to create a partial new list, map, or string value[2] This results in a high allocation rate of small heap objects making it critical that the GC can keep up with the workload without stalling.

6.1 GC Performance

The first evaluation is a throughput comparison between Pando and an ϵ-gc collector which allocates continuously from a bump buffer without any collections.

We measure the total wall-clock time taken by the application and pause times for the collector. These results are shown in Table 2. The first column is the benchmark name, the second and third columns are the total wall-clock time for the application to run with the Pando collector and the ϵ-gc collector respectively.

[2]Active engineering work is focused on improving persistent data structure implementations to reduce these issues and improve allocation and performance behavior. However, for this evaluation the extreme allocation rate provides a good test of the GC's performance under stress.

Table 1. Static and dynamic statistics for the evaluation applications. The Code Size column is lines of Bosque source code and Types is the number of distinct Bosque types in the program. The next three columns are dynamic memory statistics – AllocCount is the total number of allocations, and AllocMemory is the total bytes allocated during execution.

Benchmark	Code Size	Types	AllocCount	AllocMemory (GB)
n-body	193	68	1,248,474,177	69.5
raytracer	273	34	822,135,153	34.4
db	304	71	1,970,703,992	92.3
compiler	5120	684	1,538,395,937	140.9

Table 2. The first two columns compare the wall-clock time for executing the application under the Pando *vs.* ϵ-gc collectors. The following three columns cover the Pando collector pause time statistics at three standard measures, 50^{th} percentile, 95^{th} percentile and 99^{th} percentile.

Benchmark	Application Time (s)		Pando GC Pause (µs)		
	Pando	ϵ-gc	50%	95%	99%
n-body	1.37	1.94	31	37	51
raytracer	1.33	1.52	30	33	36
db	1.51	1.89	67	71	76
compiler	1.17	1.66	112	214	271

The results in Table 2 show that in all cases the Pando collector is actually faster than the ϵ-gc collector by on average 23%. Our analysis indicates that this is due to the improved locality of the memory access patterns after copy-compaction out of the nursery which more than offsets the additional work required by the collector. The average pause times for the collector are quite low, with a 50^{th} percentile pause time of 31 µs-112 µs across the benchmarks and a 99^{th} percentile pause time of 271 µs on any benchmark!

We also observe that the pause times are quite consistent across the benchmarks. Thus, as expected from the theoretical analysis in Section 5, we see that the collector performance is largely invariant of the application workload and primarily a function on the nursery size.

Critically, the temporal behavior seen in Table 2 is *not* achieved at the expense of memory overheads [26]. As shown in the last column of Table 3, the maximum *heap size* used by the application during the execution of the benchmark, measured as the size of all committed memory pages used in the computation, is under 12 MB for every application.

As expected from the high allocation rate in Table 1, the number of collections performed by the Pando collector is high (column 2 of Table 3) and the local/temporary nature of the allocations leads to low survival rates (column 3 of Table 3) despite the 8 MB nursery size. The GC %Time column in Table 3 is the percentage of total application time spent in garbage collection – this value is larger than is typical for a mature language/GC stack, however our analysis indicates that this is a function of an unoptimized GC codebase and high allocations rates incurred by the baseline implementations for the persistent data-structures. Despite this, the values are still all under 6.9% for all benchmarks.

6.2 GC Parameter Sensitivity

The Pando collector is designed to provide stable performance characteristics across a range of workloads and allocation patterns. However, there are fundamental configuration parameters, nursery size and page size, that critically impact the performance of the collector.

To evaluate the sensitivity of the collector to these parameters we ran a set of experiments varying the nursery size from 1 MB to 16 MB and page sizes from 2 kB to 16 kB. The results of these experiments are shown in Table 4. As our results were consistent across all benchmarks we show only the measurements for the (largest benchmark) compiler.

As shown in Table 4 the nursery size has a significant impact on the collector pause times and total percentage time spent in GC. As we would expect, smaller nursery sizes lead to lower pause times as the amount of data copied during each collection is reduced. However, this comes at the cost of increased total percentage time spent in GC as the frequency of collections increases. Values of 4 MB or 8 MB have similar performance characteristics, 4 MB has lower pause times while 8 MB has lower total percentage time in GC. Similarly, the sweep over page sizes shows 4 kB and 8 kB perform well. Based on these results, we use a default nursery size of 8 MB and a default page size of 8 kB.

6.3 Application Performance Distribution

This section presents a set of controlled experiments that are intended to evaluate the end-to-end statistical behaviors of the Bosque runtime and the Pando collector.

The first experiment examines the end-to-end latency distributions of the core benchmarks (as opposed to simply the collector latency behaviors). To evaluate this we

Table 3. The first two columns show the total number of collections performed by the Pando collector during the benchmark run and the average survival rate of the nursery (at 8 MB). The next column shows the percentage of total application time spent in GC. The final column shows the max memory used by the application, runtime, *and* collector as measured by total page usage from the OS.

Benchmark	Collections	Survival Rate	GC %Time	Heap Size (MB)
n-body	960	0.01%	2.4%	9.2
raytracer	329	0.007%	0.7%	11.4
db	1181	0.32%	5%	9.7
compiler	595	0.35%	6.9%	11.8

Table 4. Total GC percentage time and GC pause percentile times on the compiler benchmark (µs) for varying nursery sizes (1 MB-16 MB) and page sizes (2 kB-16 kB)

Nursery	Page	Total %GC Time	50% (µs)	95% (µs)	99% (µs)
8 MB	2 kB	8.4%	163	441	574
8 MB	4 kB	6.3%	125	230	286
8 MB	8 kB	6.9%	112	214	271
8 MB	16 kB	6.1%	116	255	318
1 MB	8 kB	23.8%	78	103	118
2 MB	8 kB	15%	83	111	130
4 MB	8 kB	9.3%	93	132	161
8 MB	8 kB	6.9%	112	214	271
16 MB	8 kB	4%	151	360	440

Table 5. Response percentile times (ms) for the incrementalized versions of the core benchmarks (nbody, raytracer, and db). These benchmarks run a series of small tasks, each expected with a roughly 50 ms median completion time. The next three columns show the 50th, 95th, and 99th percentile response times for each task.

Benchmark	50% (ms)	95% (ms)	99% (ms)
nbody-i	46.2	46.7	47.1
raytracer-i	51.1	51.7	52.3
db-i	50.9	51.2	51.4

created incrementalized versions of the three benchmarks, `nbody-i`, `raytracer-i`, and `db-i`, based on the corresponding core benchmarks. These incrementalized benchmarks are designed to run a series of small tasks, each expected with a roughly 50 ms median completion time, mimicking a service responding to various requests. Using these incrementalized benchmarks we can measure the end-to-end latency distributions of the applications with the Pando collector.

The results of this experiment are shown in Table 5 which shows the 50th, 95th, and 99th percentile response times for each task on over the benchmarks. As can be seen in this table the response times are tightly clustered, indicating consistent performance characteristics. The 50th percentile times for each of the core benchmarks are near 50 ms as expected while the 95th and 99th percentile time tails are very tight, with the 95th percentile being under a 2% increase over the 50th percentile times and staying under 47 ms-52 ms (less than 3% slower) even for 99th percentiles.

This is a remarkable result as it indicates that the Pando collector and runtime provide a very stable and predictable performance profile for the applications. Critically, this shows that the stability of the core GC seen in Table 2 translates into well behaved tail-latency in the end-to-end application performance as well.

Our idealized runtime model is one where the performance profile for any given task is independent from all others. Specifically, we want to examine how the distribution of response times for each operation changes when:

1. Each operation is run in isolation as a uniform workload consisting only of that operation.
2. Each operation is run as part of a mixed workload consisting of all operations intermixed.

To accomplish this we created a `Server` application that runs tasks from all three core benchmarks in a uniform worklist or in a randomly intermixed worklist, while disaggregating the timings into their respective benchmarks, via the following pseudo-code:

```
const List<NbodyTask*> nbodyTasks = ...;
const List<RaytracerTask*> raytracerTasks = ...;
```

```
const List<DbTask*> dbTasks = ...;

void recordTiming(ITask* task, GCTimeData* times) {
  if(task->isNbodyTask()) {
    nbodyTimings.record(times);
  } else if(task->isRaytracerTask()) {
    raytracerTimings.record(times);
  } else {
    dbTimings.record(times);
  }
}

void runIsolated() {
  for(size_t i = 0; i < nbodyTasks.size(); i++)
    nbodyTasks[i]->run();
  for(size_t i = 0; i < raytracerTasks.size(); i++)
    raytracerTasks[i]->run();
  for(size_t i = 0; i < dbTasks.size(); i++)
    dbTasks[i]->run();
}

void runMixed() {
  List<ITask*> allTasks = {};
  copy(nbodyTasks, allTasks);
  copy(raytracerTasks, allTasks);
  copy(dbTasks, allTasks);

  random_shuffle(allTasks);
  for(size_t i = 0; i < allTasks.size(); i++)
    allTasks[i]->run();
}

...
```

In this code snippet the `runIsolated` function runs each of the core benchmark tasks in sequence, while the `runMixed` function creates a mixed worklist of all tasks from all benchmarks and runs them in a random order. In both cases the time taken for each individual task is recorded and then disaggregated back into the timings specific to the respective benchmark task kind.

The Uniform columns in Table 6 show the average and standard deviation times for each operation when run on each benchmark workload independently. By construction these workloads are uniform and each task in the workload is expected to take roughly the same amount of time. The results in the *Uniform* columns show average response times centered around roughly 50 ms and a 2σ deviation of 0.8 ms-1.4 ms. The Mixed columns show the same metrics for the operations when run in the context of the mixed workload – that is the task is run alongside and mixed with other tasks but the times are disaggregated back into their respective tasks specific categories.

The results in the *Mixed* columns of Table 6 show that in the mixed workload the performance characteristics of individual tasks closely resemble the distributions of their uniform counterparts. Specifically the average times are only slightly higher, by 0.5 ms (under 1%), and the 2σ standard deviation is only slightly wider with a range of 1.0 ms-3.2 ms. This suggests that the mixed workload does not significantly alter the performance profile of individual tasks, supporting the notion of execution independence in our runtime model.

6.4 Direct Comparison with SoTA Java GC

The final experiment is a direct comparison between the Pando collector and modern state-of-the-art low-latency Java garbage collectors on the same benchmark. The binary-trees benchmark is a small, but widely used benchmark from the Benchmark Shootout [3], that is highly heap intensive. It is designed explicitly to stress GC algorithms by creating long-lived data structures while simultaneously allocating at a very high rate. For our purposes it is also possible to implement using exactly same code structure in both Bosque and Java allowing for a true 1-1 comparison of Bosque/Pando with an mainstream language and various heavily optimized state-of-the-art GC algorithms.

The data in Table 7 shows the results of this experiment comparing Pando with ZGC [33] and Shenandoah [9] (in their default configurations), two modern low-latency garbage collectors for Java running with the Java 25 JVM. These collectors are heavily optimized for concurrent and parallel collection, as opposed to Pando which is currently a baseline single-threaded implementation which fully pauses the application for the full collection cycle.

The first row in Table 7 shows the results when the max heap size is unlimited allowing the collectors to use as much memory (up to 64 GB) and as many threads as desired (with a 16/32 core CPU). The Overhead columns show the memory over-provisioning factor as computed by the max heap size divided by the live heap size. The Time columns show the total time taken by the benchmark in terms of wall-clock time and CPU time over all threads (possibly running in parallel). As can be seen, both ZGC and Shenandoah are able to complete the benchmark in lower wall-clock time than Pando. However, this performance comes at the cost of significant memory overheads between 8.3× and 24.5× the live heap size – in the case of ZGC using 3.2 GB when compared to 282 MB for Pando.

The second row shows the result of the benchmark with the max heap size limited to 1.5× the live heap size (300 MB). As shown in column 2 of Table 7, both ZGC and Shenandoah experience severe performance degradation under this configuration with wall-clock time 1.5×-2.0× higher and CPU time spiking by 2.1×-3.8× over their performance with unlimited heap size. Additionally, both ZGC and Shenandoah experience significant issues with GC pauses and application stalls under this configuration. The collectors report degenerate GC runs and forced synchronous collections as they are unable to keep up with the allocation rate.

Conversely, Pando experiences no performance degradation under this configuration with both wall-clock and CPU times remaining stable. The Pando continues to operate normally with increased pauses due to the higher survival rates, and heavy RC workloads, but the fundamental characteristics of the collector prevent the emergence of pathological issues. In fact removing the RC decrement phase from the

Table 6. Analysis of path-independence of operation performance – *Uniform* columns are times when tasks from a isolated workload runs are measured. *Mixed* columns are times when tasks from the mixed workload are run and then the timings disaggregated back to individual task kinds.

Benchmark	Uniform (ms)			Mixed (ms)		
	Average	1σ	2σ	Average	1σ	2σ
nbody-i	47	0.7	1.4	48.7	1.6	3.2
raytracer-i	54	0.4	0.8	53.2	1.1	2.2
db-i	53	0.4	0.8	53.7	0.5	1

Table 7. Direct comparison of Pando with state-of-the-art low-latency Java garbage collectors ZGC and Shenandoah on the binary-trees benchmark. The first column is the max heap size allowed, either unlimited or set to 1.5× the live heap size. The next set of columns shown the memory over-provisioning overhead and Wall-clock/CPU time taken by each collector – ZGC, Shenandoah, and Pando respectively.

Max Heap	ZGC		Shenandoah		Pando	
	Overhead	Wall/CPU(s)	Overhead	Wall/CPU(s)	Overhead	Wall/CPU(s)
Unlimited	24.5	2.4/4.5	8.3	2.0/2.9	1.4	5.9/6.5
1.5× Live	1.2	4.7/9.6	1.5	2.9/10.9	1.4	5.9/6.5

stop-the-world collection, *e.g.* by performing these operations concurrently on a background thread, is sufficient to keep the Pando 50^{th} percentile times at 220 µs and even the 99^{th} percentile times under 10 ms.

7 Related Work

The fields of memory management and garbage collection are vast topics [14]. Thus, in this section we focus on the most relevant aspects of these fields as they pertain to the Bosque language and Pando runtime.

7.1 The Costs of Garbage Collection

This work is heavily motivated by the analyses in [7, 26] which explore the costs, including directly incurred and visible components as well as second order effects that are diffuse but substantially impact application behavior. Recent work has shifted heavily from optimizing for throughput toward the issues of latency and application responsiveness [4, 9, 10, 33, 34]. These results enable the use of garbage collected languages in application spaces that require low-latency and would have previously had to be written in languages with manual memory management. However, as demonstrated empirically and recently theoretically [29], there are inherent trade-offs in the design space of garbage collectors that make it impossible to simultaneously optimize for both throughput, starvation, and latency in existing mainstream languages :(

7.2 Reference Counting Collectors

The idea of mixing reference counting and tracing collection in a single (generational) collector has been explored in previous work [2, 5, 6]. These approaches aim to combine the benefits of both techniques, allowing for more efficient memory management.

The integration of generations with reference counting is critical for allowing the Pando collector to avoid touching old objects (aside from the fringe) once they have been promoted. Various forms of reference-counting collectors have been explored in the literature recently [2, 5, 27, 28] and and cover many points in the design space. A key issue, as explored in [28], is the treatment of roots as precise or conservative. Although, precise roots have great appeal from a collector implementation standpoint, a conservative design presents flexibility and simplification options that are practically beneficial in many scenarios. In particular integration into larger software systems, *e.g.*. JavaScript engines [1, 21], and enabling aggressive compiler optimization without worrying about maintaining root storage invariants.

7.3 Multi-Threaded Execution

The current collector and the Pando runtime are single-threaded. However, there is active work to add structured task-parallel computation to Bosque. In this model we, ideally, want to provide fully thread-local allocation and collection. The invariant that the old space is fully reference-counted and only *fringe* objects are touched during collection, presents compelling opportunities for parallelism. Similarly, the immutability of objects and the lack of cycles in the object graph, presents opportunities for tracing concurrently with the application thread *without* the need for complex synchronization or write barriers [14].

7.4 Stack and Region Allocation

The current collector and Pando runtime could be extended to support a region-based memory management model [11, 15]. The functional nature of Bosque naturally lends itself to simpler region identification and the, already thread-local and page based, structure of the Pando collector make the implementation of stack or region allocation more practical than in a language with more complex memory semantics. However, Bosque has another feature, a focus on functor libraries [13, 17, 18] for collection processing. These libraries provide a single call for applying an operation to a `List<T>` or `Map<K, V>`.

Thus, there is also the possibility for a specialized optimizer that understands the semantics of these operations as atomic components instead of a series of individual allocations. In particular, with operation like a `map(fn)` which is of type `List<T> -> List<U>` that produces a new collection of the same cardinality and all temp values allocated in fn are dead after the call, it is possible precompute the memory needed and reduce all allocations pointer bumps!

7.5 Ownership GC

Ownership-based garbage collection is an emerging paradigm that leverages ownership semantics to manage memory more effectively. Notably, the Perceus [25, 32] collector uses the type system to detect when an object is no longer used and can be immediately recycled or efficiently updated in place. This design choice can produce very efficient executable code from the source, functional, language. However, this is a tradeoff in a system like Pando where immediate recycling of objects would break invariants around old/young object locations, *e.g.* impossibility of old→young references, and possibly introduce the need for remembered sets.

7.6 Tail Latency

Application latency, and tail-latency in particular, are critical issues in modern computing systems [8, 10, 23, 24]. The garbage collector is a critical component of a runtime system and is often a major source of variance in application performance behavior. Massive work has gone into various GC algorithms to reduce their costs – with a particular focus on latency [9, 10, 24, 33, 34]. However, in a language with mutation, cycles, and semantically observable object identity, there are fundamental limitations to what can be achieved [7, 29] – specifically tradeoffs between latency, throughput, and starvation along with the increasing complexity of the memory management implementation.

Conversely the Bosque project, and Pando runtime, present an alternative view where simplicity and simplification of the language semantics open new opportunities for garbage collection design. In particular, the results in [29] show that it is theoretically impossible for any (mainstream) imperative language to simultaneously provide low-latency and high-throughput garbage collection. This places Bosque in a unique position as the only language/runtime stack that, by allowing the assumption of very strong invariants about program state, enables the kind of aggressive garbage collection design presented in this paper.

8 Onward!

This paper presents a novel garbage collector design for the new Bosque programming language and runtime. A key design objective in this project generally, and this collector specifically, is to create a software stack that provides predictable and low-latency performance along with a very light memory footprint and small tail latencies. The Pando collector presented in this work is a key component in this system and, represents the first language/runtime/gc combination capable of satisfying the *no-tradeoff memory subsystem happiness* property (Theorem 5). The experimental results provide strong preliminary evidence that the theoretical properties of the collector are borne out in practice. As a result, we believe that this work represents a significant development in design of memory management systems for modern applications and opens up a new area of research in the design of runtime and GC systems focused on the (reliability and stability) requirements of modern software systems.

Data Availability

The system is under active development for community use with an open-source licence (MIT) via the GitHub repository at https://github.com/BosqueLanguage/BosqueCore. An archival snapshot of the code used for this paper is also registered and referenceable at [20].

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and suggestions on this work. We would also like to thank Erez Petrank for early discussions of this work as well as Ed, Curtis, Hitesh, Louis, and rest of the Chakra team for sharing their knowledge about production GC challenges!

References

[1] Apple 2025. JavaScriptCore (JSC). https://docs.webkit.org/DeepDive/JSC/JavaScriptCore.html.
[2] Hezi Azatchi and Erez Petrank. 2003. Integrating Generations with Advanced Reference Counting Garbage Collectors *(CC)*.
[3] Benchmark Shootout 2024. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
[4] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis *(ASPLOS)*.
[5] Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait *(OOPSLA)*.
[6] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance *(PLDI)*.

[7] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors *(ISPASS)*.
[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* (2013).
[9] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK *(PPPJ)*.
[10] Go GC 2024. A Guide to the Go Garbage Collector. https://tip.golang.org/doc/gc-guide.
[11] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management in Cyclone *(ISMM)*.
[12] Luke Hoban. 2008. Taking LINQ to Objects to Extremes: A fully LINQified RayTracer. https://learn.microsoft.com/en-us/archive/blogs/lukeh/taking-linq-to-objects-to-extremes-a-fully-linqified-raytracer/.
[13] Java Streams 2019. Java Streams. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.
[14] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC.
[15] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap *(PLDI)*.
[16] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. https://www.microsoft.com/en-us/research/wp-content/uploads/2019/06/mimalloc-tr-v1.pdf.
[17] LINQ 2019. LINQ. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.
[18] Mark Marron. 2023. Toward Programming Languages for Reasoning: Humans, Symbolic Systems, and AI Agents *(Onward!)*.
[19] Mark Marron. 2026. Toward an Agentic Infused Software Ecosystem. arXiv:2602.20979 https://arxiv.org/abs/2602.20979
[20] Mark Marron and Anthony Arnold. 2026. *Going Where No GC Has Gone Before! – A No-Tradeoff Memory System for Bosque*. https://doi.org/10.5281/zenodo.20182490
[21] Microsoft 2025. ChakraCore JavaScript Engine. https://github.com/chakra-core/ChakraCore.
[22] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong! *(ASPLOS)*.
[23] Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann.
[24] Tianle Qiu and Stephen M. Blackburn. 2025. Iso: Request-Private Garbage Collection. *Proceedings of the ACM on Programming Languages* (2025).
[25] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse *(PLDI)*.
[26] Kunal Sareen and Stephen Michael Blackburn. 2022. Better Understanding the Costs and Benefits of Automatic Memory Management *(MPLR)*.
[27] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the Count? Getting Reference Counting Back in the Ring *(ISMM)*.
[28] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast Conservative Garbage Collection *(OOPSLA)*.
[29] Matthew Sotoudeh. 2025. Pathological Cases for a Class of Reachability-Based Garbage Collectors *(OOSPLA)*.
[30] SPECjvm98 1999. SPECjvm98 Documentation, release 1.03 edition. https://www.spec.org/jvm98/.
[31] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence *(ICFP)*.
[32] Sebastian Ullrich and Leonardo de Moura. 2021. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming *(IFL)*.
[33] ZGC 2023. JEP 377: ZGC: A Scalable Low-Latency Garbage Collector. https://openjdk.org/jeps/377.
[34] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-Latency, High-Throughput Garbage Collection *(PLDI)*.

Received 2026-03-26; accepted 2026-05-04