Alligator Collector: A Latency-Optimized Garbage Collector for Functional Programming Languages

Ben Gamari
Well-Typed LLP
London, U.K.
ben@well-typed.com

Laura Dietz
University of New Hampshire
Durham, NH, U.S.A.
dietz@cs.unh.edu

Abstract

Modern hardware and applications require runtime systems that can operate under large-heap and low-latency requirements. For many client/server or interactive applications, reducing average and maximum pause times is more important than maximizing throughput.

The GHC Haskell runtime system version 8.10.1 offers a new latency-optimized garbage collector as an alternative to the existing throughput-optimized copying garbage collector. This paper details the latency-optimized GC design, which is a generational collector integrating GHC's existing collector and bump-pointer allocator with a non-moving collector and non-moving heap suggested by Ueno and Ohori. We provide an empirical analysis on the latency/throughput tradeoffs. We augment the established nofib micro benchmark with a response-time focused benchmark that simulates real-world applications such as LRU caches, web search, and key-value stores.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: garbage collection implementations

ACM Reference Format:
Ben Gamari and Laura Dietz. 2020. Alligator Collector: A Latency-Optimized Garbage Collector for Functional Programming Languages. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3381898.3397214

1 Introduction

A growing number of distributed systems and interactive applications require fast system response times, while the increased memory capacity of modern computers lead to a higher expected memory consumption. Language runtime systems are expected to handle large heaps while offering low latency to the mutator. Like many language runtimes, the Glasgow Haskell Compiler (GHC) uses a stop-the-world, generational, copying garbage collector [18]. While this collection strategy offers excellent memory locality, efficient bump-pointer allocation, and straightforward parallel collection, collections of the oldest generation (so-called "major collections") require to pause the mutator for durations proportional to the size of the live heap. For this reason, it is not uncommon for Haskell programs with many gigabytes of heap-managed data to exhibit pauses on the order of seconds—which is unacceptable for many applications.

We introduce Alligator, a generational mark-and-sweep garbage collector designed for the following requirements:

- Maintain predictable, fast (on the order of 10 milliseconds) pause times, even with many gigabytes of live heap-managed data.
- Provide sufficiently cheap allocations to incur minimal runtime overhead when used.
- Activated in the runtime-system without the need for recompilation.
- Portable across platforms, requiring no platform-specific virtual memory tricks.

A pure functional language like Haskell exhibits different characteristics as typical imperative programs. While mutation is ubiquitous in typical imperative programs, in functional languages, the causes are either due to thunk updates (due to lazy evaluation) or (often rare) explicitly-mutable objects such as mutable arrays and reference cells. However, functional programs tend to be very allocation-heavy, often producing gigabytes of short-lived objects per second.

GHC Haskell, unlike many high-level languages, is batch-compiled to native code and does not target a virtual machine. This complicates garbage collector design as additional barriers either must be compiled to object code, which poses deployment challenges for users, or be sufficiently fast to be generated unconditionally.

ISMM '20, June 16, 2020, London, UK

ACM ISBN 978-1-4503-7566-5/20/06...$15.00
https://doi.org/10.1145/3381898.3397214

We propose a composite garbage collector combining the existing generational copying collector of GHC for young generations with a non-moving mark-and-sweep collector for a single old generation. The existing collector allows for fast allocation through a bump-pointer allocator, which is evacuated and scavenged by copying in a stop-the-world manner. In contrast, the non-moving heap can be collected concurrently with the mutator using a mark-and-sweep approach, which facilitates short mutator pauses even when the heap is large. To facilitate safe concurrent collection we rely on the snapshot-at-the-beginning strategy of the non-moving heap coupled with generational remembered sets that ensure that both paradigms work together.

Our design allows concurrent execution of the mark-and-sweep collector with both mutators threads and the copying collector. Moreover, it requires minimal changes in mutator code, incurs minimal overhead when not in use, and can be enabled at runtime without the need for recompilation.

Contributions. We build on prior work of Ueno et al. [25] to offer the following new contributions:

- We extend Ueno's non-moving collector to a generational setting with a fast bump-pointer allocator.
- We develop an implementation that can be enabled in the runtime system at runtime without the need for recompilation.
- We offer an incremental marking scheme for evaluation stacks where mutator and mark thread share marking effort, thereby efficiently supporting a large number of deep program stacks as often arises in parallel functional programs.
- We integrate the concept of object aging and eager promotion to allow efficient treatment of thunks in our generational setting.
- We analyze the runtime behavior of our collector in comparison to GHC's existing collector, focusing on applications that require fast response times.

Outline. We review related work in Section 2 and summarize foundations for our design in Section 3. Section 4 details the design of the Alligator collector. We empirically evaluate the new collector approach in Section 5, before concluding the paper.[1]

2 Related Work

Several garbage collection approaches have been suggested to reduce the length of the pause times. These fall into two broad categories: incremental collection and concurrent collection. Incremental collectors reduce mutator interruption by interleaving GC pauses of bounded-length with mutator progress. The concurrent collectors reduce mutator interruption by allowing the bulk of collection to run concurrently with the mutator. Our Alligator collector also falls into this concurrent category.

Low-latency garbage collection in GHC. Several attempts have been made to introduce a low-latency garbage collector into GHC. An earlier attempt [6, 7] leverages an artifact of GHC's lazy-evaluation mechanism — the fact that the mutator must call a heap object's "entry" code before inspecting its contents — to implement a low-cost read-barrier. This read-barrier is used to implement an incremental tricolour garbage collection scheme [2] with low (less than 5%) runtime overhead. However, the later introduction of the "pointer tagging" optimisation [20] is incompatible with this collection approach as it eliminates mutator calls to entry code in many cases.

Orthogonally, there has also been un-merged work within GHC that explore the use of local heaps [19] to permit concurrent collection of individual mutator threads' allocation areas. While this improves the scalability of the collector by eliminating the need to synchronise across mutator threads for frequent minor collections, it does not address the long pause times caused by major collections.

While it doesn't target low latency, the Immix collector [4] explores the use of a heap structure providing allocation performance approximating that of a bump-pointer heap through use of mark-region collection.

Concurrent garbage collectors. Outside of the functional programming community many concurrent collector designs are proposed (cf. Table 1). Some are *on-the-fly* collectors, which require no pause on the mutator's part.

Doligez, Leroy, and Gonthier [10, 11] suggest a pause-free mark-and-sweep collector for sequentially consistent machines using tri-color marking and a two-phase handshake between the collector and mutator(s). Domani et al. [12] describe a similar collector for Java which includes support for use on non-sequentially consistent machines, and generational collection.

The Sapphire collector [16, 27] offers a prescription for introducing on-the-fly concurrent collection into any copying collection scheme via replication. This technique is particularly suitable for settings with high thread-counts (like many GHC programs) as the mutator can be moved one-thread-at-a-time to the replica.

The Garbage First collector [9] is a moving collector which targets soft real-time applications by focusing collection effort on heap regions which contain a high fraction of garbage. This allows the collector to enjoy the benefits of compaction while avoiding the long pause associated with collection of the entire heap. This comes at the cost of added bookkeeping to track references between heap regions.

[1] More materials and results available in the online appendix at https://github.com/well-typed/ismm-2020-nonmoving-gc/.

Table 1. Comparison of related work. The "HW reqs." column refers to hardware capabilities demanded by the collection approach: "HTM" refers to Hardware Transactional Memory and "VM" refers to non-trivial virtual memory operations which might impact portability.

Collector	Implementation	Incremental / concurrent	Moving / non-moving	Hardware requirements	Parallel collect?	Generational
Shenandoah [13]	Java (OpenJDK)	incr.	compacting		no	no
G1 [9]	Java(Hotspot JVM)	conc.	compacting		yes	yes
Ben-Yitzhak compactor [3]	Java (IBM JVM)	incr.	compacting		yes	no
Ossia compactor [21]	Java (IBM JVM)	conc.	compacting	VM	no	no
Chihuahua [1]	Java (Jikes)	conc.	compacting	HTM	no	no
Compressor [17]	Java (Jikes)	conc.	compacting	VM	yes	no
Immix [4]	Java (Jikes)	neither	non-moving		no	yes
Non-Stop Haskell [6, 7]	Haskell (GHC)	incr.	compacting		no	no
Ueno – Ohori [25]	SML (SML#)	conc.	non-moving		yes	no
Alligator (ours)	Haskell (GHC)	conc.	non-moving		yes	yes

The Shenandoah collector [13] is a concurrent compacting design targetting low pause times with very large heaps. While it relies only on the compare-and-swap operation, its reliance on a read barrier and Brooks' indirection [5] which would require recompilation.

The Compressor collector [17] is a concurrent, parallel, and incremental compaction algorithm. However, it relies on the machine's virtual memory subsystem to trap object accesses during compaction. Ossia, et al. [21] describe a similar compaction scheme intended to supplement a mark-and-sweep collection scheme. Like in Compressor, page protection is required to catch lost writes. Ossia analyses the overhead associated with page access violation events, finding that single trap events can cost more than 100 microseconds. The effective cost of these events would likely be higher with recent changes in hardware architecture and increased context-switch times due to hardware data leakage mitigations.

There have been several recent approaches [1, 23] exploring the use of the hardware transactional memory (HTM) functionality of some modern processors for garbage collection. However, Haskell supports platforms without such functionality. Also, the performance advantages are unclear [23, Section 5.2].

Alligator draws from Ueno and Ohori [25, 26], who describe a concurrent, on-the-fly collector implemented in SML#. As will be described in Section 3, we use the heap structure and snapshotting mechanism from this design, but, in the interest of simplicity, without the on-the-fly handshake. However, as our design relies on moving, stop-the-world minor collection, in practice our design stands to benefit little from on-the-fly operation.

3 Background

Since our approach heavily relies on both the Ueno nonmoving collector [25] and as GHC's existing Cheney-style copying collector [8], we will give a brief overview.

3.1 GHC's Existing Copying Collector

GHC's copying garbage collector [18] is built around an allocator (known within GHC as the "block allocator" but which refer to here as the "storage manager allocator" to avoid confusion) that offers a variety of convenient properties:

- Each block is associated with a metadata area, the *block descriptor*, which is accessible in $O(1)$ time and can be used to store information such as the generation to which the block belongs.
- While the block size is fixed, contiguous groups of blocks be allocated.
- Blocks can be linked together into lists.

On top of this allocator GHC builds a standard Cheney-style generational, copying collector. In every collection cycle, the semi-space collector evacuates live objects to *to-space*, then scavenges the freshly evacuated objects, i.e., evacuating its references and updating pointers.

Like other generational collectors, GHC's moving collector uses a write-barrier on pointer mutations to track references from objects living in old generations to objects in younger generations. This write barrier records pointers living in young generations in what we call the *generational remembered set*.

3.2 Generational Collectors

Generational collectors exploit the observation that most objects are short-lived, frequently collecting recently-allocated objects to keep residency low. Here we use the term *major collection* to refer to collection of the oldest generation and *minor collection* for collections of younger generations.

While it is common for objects in the young generation (short: *young objects*) to refer to objects in the older generation (*old objects*), references from old to objects can be introduced by mutation. While minor collections only trace the young generation, they must take care to account for such old-to-young references when determining reachability. This issue is accomplished by adding old objects referring to young objects to the root set for the minor collection. Specifically, the collection scheme must ensure that whenever a reference from an old object to a young object is introduced, the old object is added to a *generational remembered set.* This guarantee is provided via a write barrier (called the *generational write barrier*) implemented by the mutator.

3.3 Snapshot Formalism

The collector has to determine the set of reachable nodes in the heap graph while that graph is being modified by the mutator. To simplify this task, many collectors reclaim with respect to the *snapshot-at-the-beginning* reachable set [28].

The snapshot formalism enables the determination of the (conservatively approximated) set objects that were reachable at the *snapshot point* t_0, typically taken to be the beginning of collection. The difficulty lies in reconstructing the heap state at t_0 at a later time point t_1, when the heap may be concurrently mutated. To reconstruct this, the collector must maintain additional information about the heap state at t_0; herein we use the term *snapshot* to refer to this extra information.

3.4 Ueno-Ohori Non-moving Collector

Ueno et al. [25] describe a non-moving heap structure and garbage collector designed for functional programming languages such as SML#. The design accommodates concurrent and parallel collection based on a mark-and-sweep strategy that controls concurrency through a snapshot-at-the-beginning approach. While Ueno et al. introduce a fully-concurrent collection strategy, our approach is based on the single-mutator variant which we describe below.

Ueno et al. make use of a heap structure, depicted in Figure 1, which divides the memory into a set of fixed-size allocation *segments*. Each segment contains an array of fixed-size *allocation blocks* and a metadata header. Segments are aligned to the segment size, allowing efficient location of the segment header of a given block.

The segment header contains the following information: (1) AllocPtr, an allocation pointer containing the index of the first unallocated allocation block, (2) AllocPtrSnap, a snapshot pointer which preserves the value of the allocation pointer at t_0, and (3) a mark bitmap containing one bit per allocation block.

The heap consists of sub-heaps, each servicing a range of allocation request sizes. These sizes are chosen to be powers of two. Each sub-heap consists of three sets of segments: (1) a set of *filled* segments which have no empty blocks, (2) a set of *active* segments which are partially empty, and (3) one *current* segment for each mutator.

The basic operations of this heap structure are:

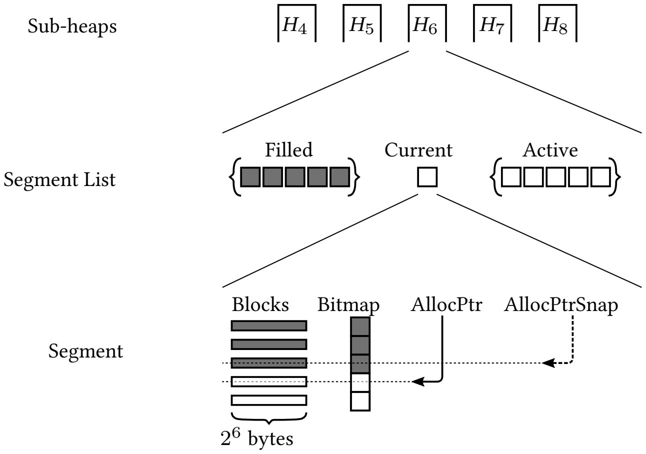

Figure 1. A schematic representation of the Ueno et al. non-moving heap structure. The heap consists of a family of power-of-two-sized sub-heaps (call-out shows 2^6 byte sub-heap), each of which consists of three sets of *segments*. Blocks occupied by (possibly live) heap objects are denoted by shading.

Allocation. The object's size determines which sub-heap services the request. The first free block, indicated by AllocPtr, is returned as the allocation result. AllocPtr is updated to point to the next unallocated block via search of the bitmap. If the segment contains no unallocated blocks, the segment is placed on the filled list for sweeping.

Write barrier. To mark all objects that were reachable at snapshot point t_0, while mutation is underway, the mutator records all overwritten pointers in the *update remembered set*, which is different from the generational remembered set described in Section 3.2.

Mark and Sweep Collection. First mark all live objects, denoted by a bit in the segments' mark bitmap. A zero in the mark bitmap means that either block is empty or contains an object that is no longer reachable. We refer to the mark stack used to track objects which have yet to be marked as WorkList.

4 Approach: Alligator Collector

The design of our generational Alligator collector is a composite of Ueno's non-moving collector for the old generation and GHC's existing copying collector for young generations. This provides several advantages:

1. The design allows us to enable the new garbage collector at runtime without requiring recompilation of the code (or any dependent libraries)[2]

[2] N.B. When GHC compiles Haskell programs to machine code, the allocation code is inlined into the mutator code, and hence cannot be easily changed at runtime.

2. Our empirical study shows that the bump-pointer allocator is much faster than other allocation schemes, which will be reflected in mutator performance.
3. In GHC's copying collector, the max pause times grow with the heap size. By keeping the moving generations at a bounded size, we restrict the max pause times even when the heap is large. The oldest generation uses Ueno's non-moving design for concurrent mark and sweep.

During a young-generation collection, the garbage collector will evacuate live data from the young generation into the non-moving heap. The Alligator collector supports use of an arbitrary number of moving young generations. However we expect that the most common configuration will involve a single moving heap for the young generation and a non-moving heap for the old generation, which is the configuration that we characterise in Section 5.

Collections of the old generation are then performed by concurrent mark and sweep. Objects in the old generation are placed in a non-moving heap. The heap design is based on the "single-mutator collector" described by Ueno [25, Section 3.1] as summarized in Section 3. Our modifications are described in Section 4.1.

The collection of the old generation, also called a major collection, is conducted in four phases (cf. Figure 2):

Preparation: Suspend mutators, collect the (moving) young generation, thereby promoting all live data into the non-moving heap, update a snapshot of the non-moving heap, enable the update write barrier, and resume mutator execution.
Marking: Concurrent marking of the non-moving heap, as described in Section 3.
Post-mark synchronization: Suspend mutators (again), mark update remembered sets, prune generational remembered set entries which were found to be unreachable, and disable the update write barrier.
Sweeping: Concurrently sweep unreachable objects from non-moving segments as described in Section 3.

Next, we provide an overview of how the two heaps and collection approaches are combined and interact.

4.1 Heap Structure

We customize the non-moving heap as follows: our non-moving allocator implementation makes use of the allocator[3] of GHC's storage manager [18] to allocate segments, allowing integration with the existing garbage collector's logic for determining object generation.

Segments are aligned block groups of 32 kilobytes in length as this provides a reasonable trade-off between heap fragmentation (which may increase with larger segment sizes) and metadata overhead (which increases with smaller segment sizes).

The Ueno allocator maintains a sub-heap for objects of each size range. In our collector we choose the sub-heap block sizes to be powers of two, ranging from 2^3 to 2^{15} bytes. Larger objects are allocated in dedicated blocks from the storage manager allocator, as is the case under GHC's existing copying collector, and are tracked separately by the collector.

4.1.1 Mark Bitmaps and State Vectors.

To maintain a high rate of allocation, the Ueno collector uses two bitmaps per segment: a tree-structured [26, Section 2.3] Bitmap to determine the next free block and TraceBitmap for marking [25, Section 4.2]. We adjust the design as follows.

We use Ueno's heap only for the old generation which exhibits a lower allocation rate than the youngest generation (which was Ueno's use case). Therefore, we forgo the complexity of this tree-structured bitmap in our generational collector. In contrast, we propose to use a smaller flat representation which combines Bitmap and TraceBitmap.

We further adjust the design to enable efficient parallel marking. In Ueno's algorithm the entries of Bitmap are modified during two stages of collection: (a) clearing of mark bitmap when a filled segment is prepared for marking, and (b) individually setting a block's mark flags as each block is marked. Parallel marking requires that mark threads do not race when setting mark flags of two objects that are placed adjacently in the heap. Since marking is idempotent, we avoid the runtime penalty of atomic memory operations by representing the bit state as a byte, since writes to bytes are guaranteed to be non-interfering on all supported architectures. We call the bitmap when represented as an array of bytes the segment's *state vector*.

We avoid the need to clear the bitmaps before marking, which would incur $O(\text{heap size})$ cost, by using globally alternating *mark epochs* (referred to as epoch A and epoch B). Each block's state byte may take one of three states: *empty* (represented by 0) for unoccupied blocks, *epoch A* (1), indicating that the block is occupied by an object which was last marked in epoch A, or epoch B (2). The global mark epoch is advanced at the beginning of every major collection cycle, and will be used during marking and sweeping.

Replacing the mark bit with a state byte increases the total space requirement only marginally: Assuming a worst case of a heap filled with small heap objects of the minimum block size of 2^4 bytes, the state vector overhead is 6%. In real-world applications with larger objects the bitmap overhead is observed to be about 2.5%.

4.1.2 Scavenging Freshly-Promoted Objects.

Ueno's non-moving collector design did not foresee use in a generational setting. Therefore, the garbage collection algorithm requires that all objects are maintained by the non-moving

[3] GHC refers to its allocator as the "block allocator." However, for the purposes of this paper we will refer to it as the "storage manager allocator" to avoid confusion with "block" used by Ueno's heap structure.

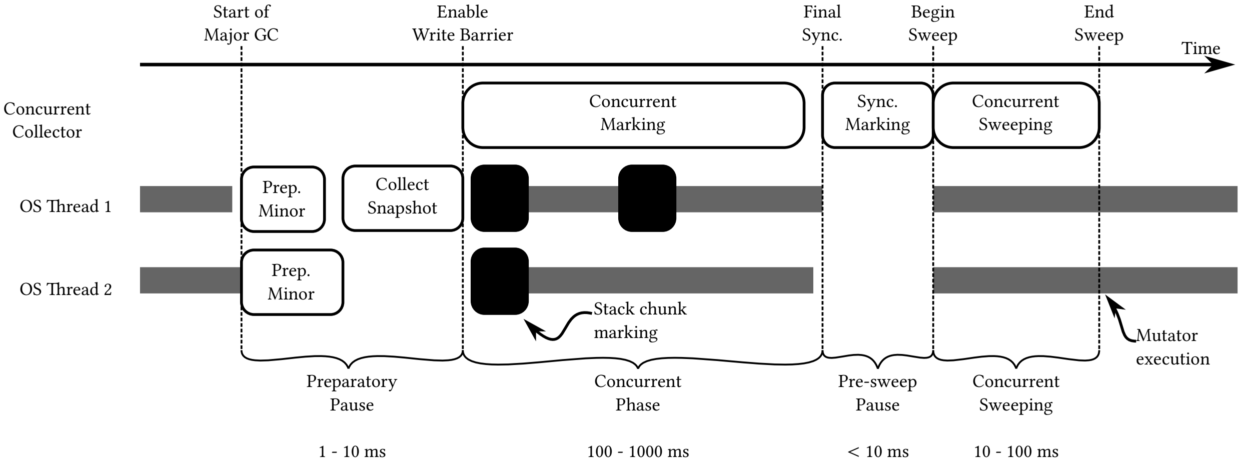

Figure 2. The lifecycle of a typical major garbage collection under Alligator. The timings are typical figures from a program with a few gigabytes of data in the non-moving heap on a modern x86-64 machine.

heap. Hence, a major collection would require to first evacuate all live data in the young generation to the non-moving heap. However, this incurs an unacceptable runtime cost as large numbers of short-lived objects would need to be evacuated to the non-moving heap whenever a major collection is performed. This is undesirable for two reasons: (a) evacuation into the non-moving heap is more expensive than evacuation into the moving heap due to higher allocation costs, and (b) this short-lived data will likely be retained in the old generation long after it has become unreachable, unnecessarily growing the program's residency. We suggest a better approach in Section 4.5, allowing the collector to choose to retain some objects in the moving heap (commonly known as *aging*).

A minor collection will promote young objects from the moving heap into the non-moving heap. GHC's existing collector is a typical Cheney-style collector, consisting of two operations: first a set of objects are *evacuated* into *to-space*; then freshly-evacuated objects will be *scavenged* by evacuating their referenced objects and updating references accordingly. The scavenging process requires an inexpensive means of identifying freshly-evacuated objects in need of scavenging. A bump-pointer allocator would place freshly-evacuated objects contiguously in memory; Cheney's algorithm exploits this by requiring only book-keeping of the begin and end pointer of the region of *to-space* which was evacuated into (known as the *scan* and *limit* pointers).

However, it is not straightforward to identify freshly-evacuated objects in the non-moving heap as these objects are interspersed with pre-existing allocations. While one solution would be to maintain a list containing each individual object in need of scavenging, this would come at a significant cost.

A simpler solution is to use Cheney-style scan and limit pointers and keep track of which objects in that objects in that range are freshly-evacuated via the segment's state vector: If a block's state is empty (0), we know that the object in this block was freshly evacuated. Since freshly-evacuated objects are allocated into a CURRENT segment, the bitmap of which is not touched by the concurrent mark phase, the allocator can safely rely on the bitmap to find free blocks. Note that we must avoid scavenging objects that are not freshly evacuated. Since their liveness is unknown they could contain references to already-swept objects.

4.2 Write Barrier for Update Remembered Set

If programs would not require mutation, our design so far would guarantee that objects in the old generation do not hold references to the young generation, and hence major collections of the non-moving heap could be performed in parallel to minor copying collections without any potential interaction. In this section we describe how the collector supports mutation.

Mutation affects our design is by introducing pointers from old objects to young objects. These old objects will be added to the generational remembered set to act as additional roots during minor collections (cf. Section 3).

To support concurrent moving and non-moving collections, we need to ensure the correctness of the reachability analysis for old objects O' that are only reachable from young objects Y, especially if those are only reachable from old objects O (depicted in Figure 4). Specifically, because the minor collection would move live young object Y (thereby temporarily invalidating the pointer in O), we must take care not to trace objects in the young generation. This is addressed by promoting all young objects Y at the beginning of the non-moving collection (with the exception of aging, see Section 4.5).

The snapshot invariant of the non-moving collector guarantees that any objects promoted to the non-moving heap after the snapshot point t_0 will not be affected by non-moving collection. However, because references within the old generation may be overwritten, we need to ensure that that all objects that were reachable at the snapshot point t_0 will be correctly marked. For example, if at time point $t_1 > t_0$, a reference from an old object O to an old object O' is mutated to instead point to object X, we need to ensure that O' is marked to respect the snapshot invariant. However if marking would inspect object O after time t_1, it would instead proceed to trace X. While it is unnecessary (but tolerable) to mark newly introduced objects such as X, we need to ensure that the old referent O' is also added to WorkList. We implement this by collecting such objects O', whose referees are overwritten by mutation, in an *update remembered set* which feeds into the WorkList. We enable a write barrier for old objects to populate the update remembered set. Unlike the generational write barrier, the update write barrier only needs to be enabled during the concurrent mark phase of collection.

In pure functional programming languages, only a limited number of causes can lead to mutation. In GHC Haskell, pointer mutations arise from three sources:

- Advancement of the evaluation stack,
- Thunk updates, and
- Modification of mutable arrays and reference cells.

Figure 3 presents the pseudo-code for the write barrier on pointer updates. This write barrier incurs two branches only while non-moving marking is underway. The code-size overhead of the barrier is negligible as most mutation code (e.g., thunk updates and most mutable object mutation) is defined in the runtime system, not the compiler-generated mutator code.

The update remembered set, like the generational remembered set, is accumulated in a per-thread data structure to minimize contention in parallel programs. During a non-moving collection three events may prompt a thread to flush its local accumulator set to the global WorkList:

- The local accumulator is filled,
- The concurrent mark phase ends, and a global sync is requested,
- The program performs a minor collection.

While flushing the update remembered set on minor GC is not necessary for correctness, it limits the size of the local accumulator and thereby reduces the amount of marking that must be performed during the final sync phase of collection, reducing pause times.

4.3 Marking Stacks

The GHC runtime system accommodates very deep evaluation stacks. Evaluation stacks in GHC Haskell are organized in a chain of stack chunks of variable size. To the collector, a stack chunk looks like any other heap object, and hence may live in the non-moving generation. Like all heap objects, the non-moving collector must ensure that all references reachable through evaluation stacks at t_0 will be traced.

```
 // Mutate pointer array with new value.
void WriteArray(Array *arr, int i, Object *value) {
      if (update_write_barrier_enabled
            && IsInOldGeneration(arr))
         push arr[i] to UpdateRemembSet;
      arr[i] = value;
}

// Overwrite thunk object with result of evaluation.
void UpdateThunk(Thunk *thunk, Object *result) {
      overwrite thunk with indirection pointer to result;
      if (IsInOldGeneration(thunk)) {
         if (update_write_barrier_enabled) {
            for (Object *p in free variables of thunk)
               push p to UpdateRemembSet;
         }
         push thunk to GenRemembSet;
      }
}

// Update IORef mutable reference cell.
void UpdateIORef(IORef *ref, Object *new) {
      if (! IsMarkedDirty(ref)) {
         if (update_write_barrier_enabled)
            push ref.value to UpdateRemembSet;
         push ref to GenRemembSet;
         MarkDirty(ref);
      }
      ref.value = new;
}
```

Figure 3. Pseudo-code for write barrier used on array writes, thunk, and mutable reference cell updates. The barriers on other mutable objects are similar to the array case. Global flags are denoted in snake-case, where functions, variables and types are denoted in camel-case.

Since stack operations are extremely frequent, tracking overwritten references through a write barrier would incur a large overhead on mutator execution. Instead, we implement a special marking behavior for evaluation stack chunks which guarantees that a thread does not resume execution on a stack chunk until that chunk has been marked.

When a mutator starts the execution on a new stack chunk, it checks whether the stack chunk has been marked (using the mark byte in the state vector): If it is not yet marked, the thread walks the stack chunk, pushing all pointers to its update remembered set. In either case, the thread proceeds with the execution. As depicted in Figure 2, stack chunk marking and mutation can be interleaved. Moreover stacks can be marked by the concurrent mark thread, whichever happens first.

This introduces a potential race as mutator and concurrent mark thread may simultaneously attempt to mark the same stack. To avoid this, the stack must be locked before initiating marking. If the mutator fails to lock the stack it waits until the concurrent mark thread has finished marking before commencing with execution. If the concurrent mark thread cannot acquire the lock, it defers marking of this chunk to the mutator and will proceed with the next item on WorkList.

As stack chunks are of bounded size (32 kBytes), the amount of work necessary to mark a stack chunk prior to execution is bounded. This approach complements the stack dirtying logic already present for the existing moving collector, which ensures that all stacks are pushed to the generational remembered set prior to mutation.

4.4 Write Barrier for Reference Cells

GHC Haskell exposes a few types of primitive mutable reference cell: MVar, IORef, and TVar. These cells contain a single reference to a heap object and represent a significant source of mutation in Haskell programs. Consequently, both GHC's existing and Alligator collectors focus on minimizing the barrier costs for these objects.

Each mutable reference cell has a single-bit "dirty" flag which is set when the object is mutated (and hence added to the generational remembered set). This allows future mutations to avoid repeatedly adding the object to this remembered set. The dirty flag is reset during scavenging if the object does not hold references into younger generations.

The non-moving collector employs a similar technique to avoid added update remembered set traffic, taking advantage of the fact that the non-moving write barrier is only obligated to push the pointer value present at t_0, and not those arising from subsequent mutations.

Specifically, the write barrier will push the overwritten pointer value to the update remembered set only if the object is clean (implying that this is the first mutation to the object since the last GC). See Figure 3 for code of this barrier.

For this to be safe we must ensure that all reference cells referring to objects in the non-moving generation are either (a) marked as clean, or (b) have their referents at t_0 on the mark worklist. To accomplish this, we scavenge any reference cells on the generational remembered set during the preparation phase of collection and mark each cell as clean if it contains no references into the young generation, otherwise (e.g. in the case of an aged object, as we will see below) push its referent to the update remembered set.

4.5 Eager Promotion and Aging

So far, we explained our design with the assumption that major collections would first evacuate all live data to the non-moving heap, as this ensures that the concurrent mark can safely trace all live data. However, due to mutation, some objects such as evaluation stacks and mutable objects are associated with many short-lived objects. We would like to avoid promoting short-lived objects to the non-moving heap for several reasons: (1) allocation in the non-moving heap is much slower than using the bump-pointer allocator of the young generation. (2) The young generation is collected more frequently, where unreachable short-lived objects in the old generation consume unnecessary heap space.

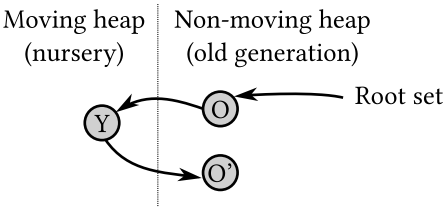

Figure 4. Handling inter-generational references during concurrent collection with aging. However, allowing Y to remain in the young generation during preparatory GC may hide live object O' from mark.

For these reasons, generational collectors introduce *object aging*, i.e., where some objects, such as evaluation stacks and mutable arrays, remain in the young generation instead of being promoted, as depicted in Figure 4.

However, aging complicates the concurrent collection approach, since we cannot safely examine young objects, because the copying collector may be in the process of evacuating them. Instead, we have to ensure that objects in the non-moving heap which are reachable only via aged objects are marked via some other means.

Whenever an object is aged by keeping it in the young generation during evacuation, all non-moving heap object references encountered during scavenging are placed on the mark work list. Aging is applied to objects which are likely to be subject to frequent mutation including STM transaction records, mutable reference cells (e.g. IORefs), mutable pointer arrays, and a variety of runtime-internal bookkeeping structures (e.g., blocking queues).

Before our Alligator collector accommodated aging, we observed unacceptably high increases in bytes copied by the non-moving collector.

4.6 Other Issues

GHC's garbage collector is used for detecting threads that are deadlocked due to blocking on un a concurrency primitive (e.g. MVar and TVar) that is not reachable.

Providing this functionality in the Alligator collector is complicated by the conservative treatment of inter-generational reference cycles that arise due to aging. In particular, consider the scenario depicted in Figure 4 with an additional edge from O' to O (inducing a cycle). If O is not reachable from the root set, we would want to identify a deadlock. However when Y is aged, O remains on the generational remembered set, and we would consider the entire cycle to be reachable, which would hide the deadlock. To avoid this failure of deadlock detection, we use a heuristic to disallow aging when there there are no runnable threads (and we therefore suspect a deadlock).

5 Evaluation

The Alligator garbage collector is implemented in the GHC Haskell runtime system, available in GHC 8.10.1.[4] A primary goal of the Alligator collector is to offer consistent short maximum GC pause times. While the previous moving collector was optimized for throughput, its stop-the-world nature potentially produced long pause times which would be unacceptable for many latency-sensitive applications, particularly in the presence of large heaps.

To substantiate our claims of response-time improvements in our GC architecture, we characterise the Alligator collector in comparison to the original throughput-optimized copying collector on several benchmarks.

5.1 Evaluation Paradigm

Unless otherwise noted, all experiments are taken on a AMD Ryzen 2990WX 32-core processor with 512KB L2 cache and 64GB of RAM, running Debian 10.

5.1.1 Benchmarks. Our evaluation characterises a selection of Haskell programs which seek to model a wide range of program behaviors (due to space, we report only a subset of results in the paper; additional benchmark results can be found in the online appendix). We briefly summarise these programs below:

nofib: Nofib [22] is a benchmark suite consisting of several dozen small Haskell programs. It is included for the sake of completeness as it is commonly used as a timing benchmark in the functional programming community. However, we note that very few of these benchmarks serve as useful benchmarks of the Alligator collector as the maximum heap residencies (and, consequently, GC pause times) of these testcases are generally quite small. All benchmarks are compiled with "-O1" and are run in their "slow" configuration.
lru-cache: Simulates a typical caching HTTP server application. The application loads a large static data structure into the heap and proceeds to serve HTTP requests. Each request looks up the query string in an LRU cache to see whether a cached result is available. If so, it is returned; if not, the result is computed, inserted into the LRU cache, and the cache pruned to maintain a size of 640,000 entries. The query results are Wikipedia pages rendered to HTML from their TREC CAR representation [14].
Load is provided by the wkr2 load generator simulating requests a one-hour-long Wikipedia query log from 2016. To ensure a balance of cache misses, the request distribution is "flattened" by clamping request count at 1000.
text-search: Incrementally builds an inverted index for web search applications from small batches of user-provided documents. Retrieval of top-k rankings of keyword-based search queries, using a term-frequency-based ranking function similar to well-known retrieval models TF-IDF or BM25. Requests are of two types: *add document* requests cause the server to incrementally build an inverted index for a named document and merge the index into the global index state. *Query* requests use the global index state to perform a top-k query for a given set of query keywords. The queries and documents are based on Wikipedia titles and passages, which are taken from the TREC Complex Answer Retrieval document corpus [14].
edit-distance: Computation of the Levenshtein distance between character strings using a dynamic programming algorithm implemented via memoized recursion. Each request requires the computation of a all pairs in a set of 18 input strings of 50 characters. The strings are taken from Wikipedia abstracts.
kv-store: Models an in-memory database server application by building and querying a large on-heap dictionary, recording the service-time of each request. Both keys and values are random integers and the dictionary is provided by the size-balanced binary tree implementation of the containers library.
list-alloc: This program is a microbenchmark which measures allocation cost by allocating a linked-list of Ints of length 10^8.
ghc-cabal: Simulates a large complex application without response-time critical nature. We use GHC, which is itself written in Haskell and hence runs on the Haskell GHC runtime system, to compile Cabal, a Haskell library of moderate size.

Test cases lru-cache and text-search use Wikipedia data provided as part of the TREC Complex Answer Retrieval competition [14]. The dataset is derived from Wikipedia to offer a corpus de-duplicated content from Wikipedia. The dataset is available under CC-SA and was produced in cooperation with the National Institute of Standards and Technology who organizes the Text Retrieval Conference, a series of shared tasks in information retrieval.

5.1.2 Evaluation Measures. We evaluate by the following measures of response-time in addition to overall elapsed runtime:

Response latency is characterised for the lru-cache benchmark with the wrk2 HTTP load generator and measures the time between the sending of a request and receiving the reply.[5]

[4] Activated with the -xn runtime system flag.

[5] The wrk2 HTTP load generator is available at https://github.com/giltene/wrk2. The generated load is exhibits a constant request rate, thus avoiding the problem of coordinated omission[24].

GC pause times are measured by the GHC runtime system. We examine maximum pause time, average pause time, and pause time histogram for both moving and non-moving collections.

Service times are measured for the lru-cache, inv-index, and kv-store benchmarks as wall-clock time to compute each work item.

Elapsed runtime is measured as total wall-clock time from the start to the exit of a benchmark process.

5.2 Experiment 1: Response and Pause Times

Many applications require consistently short response-times, such as servicing requests in a client-server fashion, and interactive applications such as games. In such settings it is unacceptable for the program to be unresponsive due to a long garbage collection pause. In this first experiment, we analyse the population of response and pause times through the course of many requests.

Figure 5 shows the distribution of response times under two server configurations: the *A* configuration measures the collector at a one core with a request rate of 2000 requests per second, while the *B* configuration uses a high core-count and higher request rate (eight cores with 8000 requests per second). For each configuration the load generator configuration was chosen such that it avoided saturation and was roughly proportional to the server core count.

We observe a long tail of response times on the order of one second for the moving GC. In contrast, the Alligator collector's tail converges to approximately 20 milliseconds. We also note that the two collectors exhibit qualitatively different behavior as the server core count is increased: While the moving collector's tail latency decreased when the core count was increased, the tail latency of the non-moving collector increased.

We complement this experiment with an analysis of the max and average garbage collection pause times, shown in Figure 6. Here we see that the response latency distribution reflects that of garbage collection pauses.

5.3 Experiment 2: Throughput of Non-moving Collector

To evaluate the throughput of the non-moving collector, we look at elapsed runtime for each of the experiments, summarised in Table 2. We will discuss two benchmarks in particular: ghc-cabal and nofib. GHC is used as a typical example of a large-scale Haskell program, requiring a moderate heap size and using a mix of laziness, explicit mutation, and optimised computation proportionate to what one would find in a typical Haskell program. The nofib testsuite is used as a worst-case for the collector as residencies are small and tests tend to be laziness- and allocation-heavy. For all of these tests the elapsed runtime and mutator time are measured.

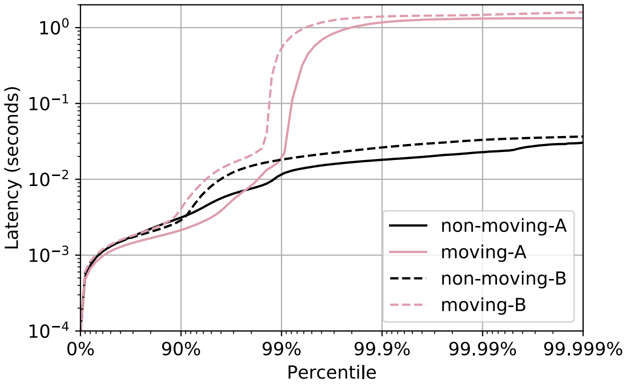

Figure 5. Response time distribution for the lru-cache benchmark run in four configurations: non-moving-A and moving-A run with one operating system thread and a load of 2000 requests per second. non-moving-B and moving-B run with eight operating system threads and serve 8000 requests per second. The peak residency of the server is approximately 8 gigabytes.

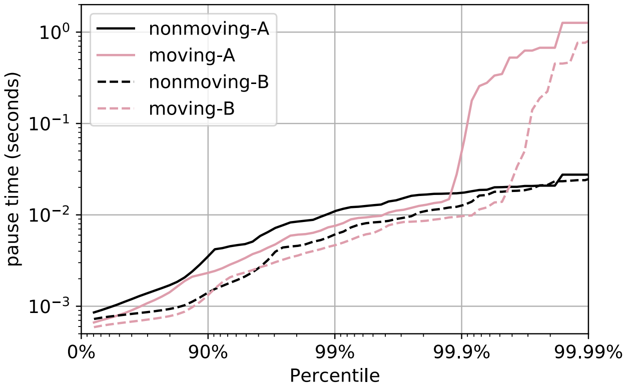

Figure 6. Garbage collector pause time distribution for the lru-cache benchmark run.

The nofib cases are quite mixed. While a quarter of tests' runtimes are unchanged or slightly improved (the gc/mutstore1 test improves by 11%), most tests slow down, with a median of +21%. One test, spectral/ansi increases in runtime by 450%, exhibiting an increase in bytes copied by a factor of 20. This result is consistent with a failure to age a heavily mutated heap object, resulting in the promotion of large quantities of short-lived garbage.

The ghc-cabal benchmark, which is more realistic, is significantly more favorable. In particular, total wall clock time increases by a mere 5%, from 176 seconds to 187 seconds whereas mutator time increases from 143 seconds to 156 seconds. The ghc-cabal benchmark also reveals an interesting pause behavior: the program exhibits a single long pause of 550 milliseconds due to a moving collection, by contrast

Table 2. Summary of runtime characteristics of the latency-oriented benchmarks when run under the Alligator collector. All percentages are relative to the copying collector. "avg." denotes arithmetic mean.

test		text-search			kv-store			lru-cache			edit-distance	
collector	*copy*	*ours*	*%chg*	*copy*	*ours*	*%chg*	*copy*	*ours*	*%chg*	*copy*	*ours*	*%chg*
metric												
elapsed time	193.2 s	229.8 s	+19%	65.9 s	72.8 s	+11%	313.5 s	311.7 s	−1%	86.9 s	69.6 s	−20%
CPU time	193.1 s	374.5 s	+94%	65.8 s	103.3 s	+57%	1493.7 s	1481.8 s	−1%	86.8 s	112.0 s	+29%
avg. min. GC pause	235.0 μs	390.0 μs	+66%	284.0 μs	466.0 μs	+64%	785.0 μs	1.0 ms	+29%	342.0 μs	824.0 μs	+141%
max. min. GC pause	4.3 ms	10.0 ms	+133%	3.2 ms	3.3 ms	+3%	14.1 ms	74.4 ms	+428%	3.2 ms	6.6 ms	+103%
avg. maj. GC pause	680.6 ms	3.5 ms	−99%	102.8 ms	557.0 μs	−99%	363.3 ms	8.1 ms	−98%	2.8 s	1.5 ms	−100%
max. maj. GC pause	4.1 s	31.4 ms	−99%	457.4 ms	725.0 μs	−100%	1.3 s	32.2 ms	−98%	20.1 s	5.7 ms	−100%

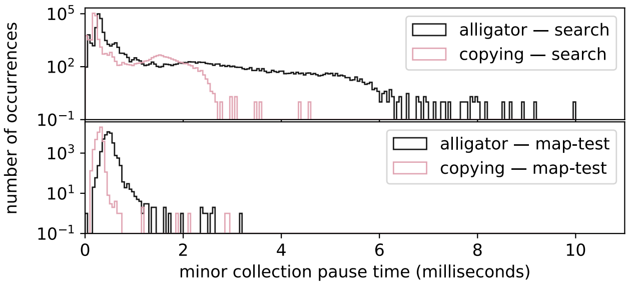

Figure 7. Distribution of minor collection pause times for runs of the kv-store and search benchmarks.

the remaining moving collection with an average collection pause of 9 milliseconds. We believe that this ourlier is due to a thunk leak within GHC which results in a large quantity of thunks being added to the generational remembered set, which contributes a large number of roots to the moving collection.

Another mechanism by which the non-moving collector might affect overall throughput is the increased overhead associated with allocation in during minor collection. To characterise this we examine the minor collection pause of the list-alloc benchmark under the moving and non-moving collectors. Due to the allocation-heavy nature of this benchmark, the cost of minor collections will be dominated by the cost of allocation. This is confirmed by the CPU profile, which reveals that over 10% of CPU time is spent in the non-moving allocator. The minor collection pause distribution reveals that minor pauses grow by a factor of three relative to the moving collector's bump-pointer allocator.

In Figure 7 we compare minor GC pause time distributions of the kv-store and search benchmarks. Here we see that the non-moving allocator reduces minor collector throughput by between 30% and 50%.

5.4 Experiment 3: Heap Fragmentation

To simplify concurrent collection, the Alligator collector makes the intentional trade-off of only sweeping filled segments. While this choice will inevitably result in some fragmentation of the heap, we hypothesize that typical Haskell programs have consistent enough allocation behavior that partially-filled segments will quickly be filled and collected.

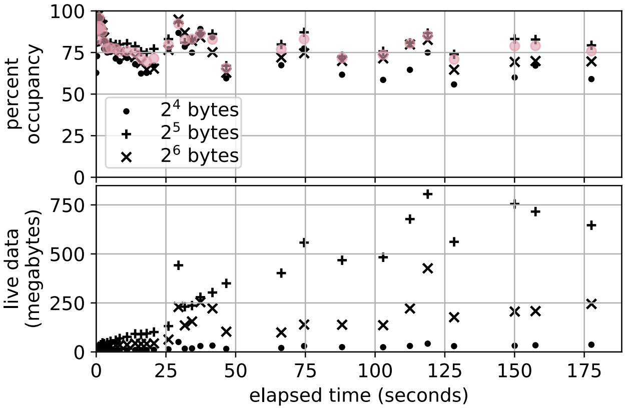

Figure 8. The top pane shows the evolution of non-moving heap occupancy over time for an execution of the ghc-cabal benchmark. The series correspond to the three smallest sub-heaps by block size, having blocks sizes of 16, 32, and 64 bytes. The red points show the overall occupancy across all sub-heaps. The bottom pane shows the quantity of live data in each of these sub-heaps over time.

To characterise this we examine the *heap occupancy* which we define to be the fraction of blocks in the non-moving heap which contain live data. Figure 8 shows the evolution of the heap occupancy for the three smallest block-sizes from a typical execution of the ghc-cabal benchmark. Note that Haskell heaps are in large part dominated by small allocations: the 2^5 and 2^6-byte sub-heaps comprise nearly 80% of the heap size for the majority of the program's execution.

Figure 8 confirms that for this typical workload the rate of allocation is sufficient to ensure that the heap occupancy remains between 70% and 90%. This is an acceptable range for most applications, especially when compared to the baseline of the copying collector which doubles the program's residency during evacuation.

We note that at around $t = 115$ seconds we observe an increase in live data across all sub-heaps. This correlates with the long pause in Section 5.3.

5.5 Experiment 4: Proprietary Server

In addition to the benchmarks described above, the Alligator collector is tested via a small test run of a proprietary server

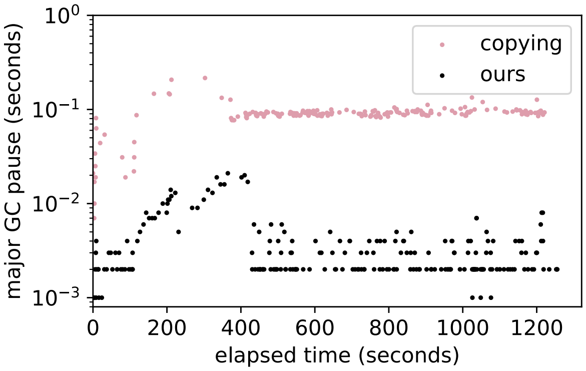

Figure 9. Progression of major GC pause times from a run of a proprietary server application using the copying and Alligator collectors.

application in use by a commercial client to minimize collection pauses in an algorithmic trade quoting service. The service ingests XML trades and numeric market data and calls a C++ library to perform a quantitative analysis of the trade, producing a bid/ask quote among other measures. In this small test the server has a peak residency of a bit more than 100 MBytes. Figure 9 shows the evolution of major GC pause times over the course of the run.

Pause times rise during the the initialization phase of the program (from $t = 0$ to $t = 400$ seconds) and eventually stabilize once the program enters its serving loop. The difference between the copying and Alligator collectors is stark, with the latter routinely exhibiting 50-fold lower pause times than the latter, even with this relatively small heap.

5.6 Experiment 5: Qualitative Analysis of Frag

To get a qualitative impression of the performance differenced between the copying collector and the new Alligator collector in an inter-active real-time system, we played the game frag,[6] a Doom-like first-person shooter with 3D rendering on a dual-core 2.66 GHz CPU with 32GB of RAM.

The performance impression difference is quite drastic: In the existing copying collector many frames are dropped, given a ragged and nauseating impression of the game. When played for 15 seconds, we observe pause times are 42 ms at maximum and 21 ms on average.

By contrast the game play is smoother and responsive with the Alligator collector, which is explained by much shorter pause times with a maximum pause of 8 ms and average of 1 ms. Hence, we conclude that with the Alligator collector, GHC Haskell is becoming suitable for interactive applications.

6 Conclusion

The Alligator collector is a composite moving/non-moving garbage collector that combines the advantages of fast young generation allocation through a bump-pointer allocator with the short pause times of a concurrent non-moving collector. The joint marking of program stacks by mutator or mark threads allows efficient handling of the large thread counts present in typical Haskell server applications while the write barrier design exploits Haskell's pure nature to provide efficient concurrent collection without the need for recompilation.

Our empirical evaluation demonstrates that pause times are reduced by a factor of 10 to 100. Accordingly, the response-times (latency) of the 99.9 percentile of the lru-cache benchmark improved from 1.3 seconds in the copying collector to 20 milliseconds under the Alligator collector.

Significant increases in allocation costs are exhibited by some benchmark programs, as is clearly demonstrated in the list-alloc benchmark. This is in part the result of our choice of allocator, which makes little effort to optimise bitmap scans. The hierarchical bitmap structure described by Ueno [26] presents a clear avenue for improvement.

In the course of our evaluation we encountered a few programs where the concurrent collector was running for a large fraction of the program's overall runtime. While the mark FIFO [15] employed by the collector contributes a modest improvement, mark performance is still heavily constrained by cache locality. We wonder whether consolidating marking work via batching along segment lines might provide better locality and consequently marking performance.

We demonstrate that even for the high allocation-rate mutators of functional programming languages, a concurrent collector can achieve competitive pause times. This is consistent with work on collection for Java [13] and Go. While on-the-fly collectors are well-represented in the literature, our evaluation shows that a simple concurrent collector can yield acceptable pause times for typical server applications. In our view, the most fruitful avenues for future improvements lie in improving non-moving allocator throughput and marking performance.

Acknowledgments

We would like to thank Ömer Sinan Ağacan, whose code, skillful debugging, and helpful discussions during the implementation part of the project were invaluable in bringing it into its current state.

We would also like to acknowledge the contributions of Pepe Iborra and Atze Djikstra whose assistance in testing and characterising the collector.

References

[1] Todd A Anderson, Melissa O'Neill, and John Sarracino. 2015. Chihuahua: A concurrent, moving, garbage collector using transactional

[6] Frag is available at https://wiki.haskell.org/Frag.

memory. *TRANSACT* (2015).

[2] Henry G Baker Jr. 1978. List processing in real time on a serial computer. *Commun. ACM* 21, 4 (1978), 280–294.

[3] Ori Ben-Yitzhak, Irit Goft, Elliot K Kolodner, Kean Kuiper, and Victor Leikehman. 2002. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd International Symposium on Memory Management*. ACM New York, NY, USA, 100–105.

[4] Stephen M Blackburn and Kathryn S McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *ACM SIGPLAN Notices* 43, 6 (2008), 22–32.

[5] Rodney A Brooks. 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 256–262.

[6] Andrew M Cheadle, Anthony J Field, Simon Marlow, SL Peyton Jones, and R Lyndon While. 2000. Non-stop Haskell. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional programming*. 257–267.

[7] Andrew M Cheadle, Anthony J Field, Simon Marlow, SL Peyton Jones, and R Lyndon While. 2004. Exploring the Barrier to Entry: Incremental Generational Garbage Collection for Haskell. In *Proceedings of the 4th International Symposium on Memory Management*. 163–174.

[8] Chris J Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (1970), 677–678.

[9] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[10] D. Doligez and G. Gonthier. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*. ACM New York, NY, USA, 113–123.

[11] D. Doligez and X. Leroy. 1993. A concurrent generational garbage collector for a multi-threaded implementation of MR. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*. ACM New York, NY, USA, 113–123.

[12] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. 2000. A Generational On-the-Fly Garbage Collector for Java. *SIGPLAN Not.* 35, 5 (05 2000), 274–284. https://doi.org/10.1145/358438.349336

[13] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–9.

[14] Ben Gamari and Laura Dietz. 2019. TREC CAR 2.3: A Data Set for Complex Answer Retrieval. http://trec-car.cs.unh.edu/

[15] Robin Garner, Stephen M Blackburn, and Daniel Frampton. 2007. Effective prefetch for mark-sweep garbage collection. In *Proceedings of the 6th international symposium on Memory management*. 43–54.

[16] Richard L Hudson and J Eliot B Moss. 2001. Sapphire: Copying GC without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. 48–57.

[17] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM New York, NY, USA, 354–363.

[18] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. 2008. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th ACM SIGPLAN International Symposium on Memory Management*. 11–20.

[19] Simon Marlow and Simon Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management*. ACM New York, NY, USA.

[20] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. 2007. Faster laziness using dynamic pointer tagging. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* 42, 9 (2007), 277–288.

[21] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. 2004. Mostly concurrent compaction for mark-sweep GC. In *Proceedings of the 4th International Symposium on Memory Management*. ACM New York, NY, USA, 25–36.

[22] Will Partain. 1993. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*. Springer, 195–202.

[23] Carl G Ritson, Tomoharu Ugawa, and Richard E Jones. 2014. Exploring garbage collection with Haswell hardware transactional memory. In *Proceedings of the 2015 International Symposium on Memory Management*. ACM New York, NY, USA, 105–115.

[24] Gil Tene. 2016. How not to measure latency. https://www.infoq.com/presentations/latency-response-time

[25] Katsuhiro Ueno and Atsushi Ohori. 2016. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 421–433.

[26] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. 2011. An efficient non-moving garbage collector for functional languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ACM New York, NY, USA, 196–208.

[27] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. 2018. Transactional Sapphire: Lessons in High Performance, On-the-fly Garbage Collection. *ACM Transactions on Programming Languages and Systems* 40, 4 (December 2018). https://kar.kent.ac.uk/67207/

[28] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181–198.