Garbage Collection for Mostly Serialized Heaps

Chaitanya S. Koparkar
Indiana University
Bloomington, USA
ckoparka@indiana.edu

Vidush Singhal
Purdue University
West Lafayette, USA
singhav@purdue.edu

Aditya Gupta
Purdue University
West Lafayette, USA
gupta782@purdue.edu

Mike Rainey
Carnegie Mellon University
Pittsburgh, USA
me@mike-rainey.site

Michael Vollmer
University of Kent
Canterbury, United Kingdom
M.Vollmer@kent.ac.uk

Artem Pelenitsyn
Purdue University
West Lafayette, USA
apelenit@purdue.edu

Sam Tobin-Hochstadt
Indiana University
Bloomington, USA
samth@cs.indiana.edu

Milind Kulkarni
Purdue University
West Lafayette, USA
milind@purdue.edu

Ryan R. Newton
Purdue University
West Lafayette, USA
rrnewton@purdue.edu

Abstract

Over the years, traditional tracing garbage collectors have accumulated assumptions that may not hold in new language designs. For instance, we usually assume that run-time objects do not hold addressable sub-parts and have a size of at least one pointer. These fail in systems striving to eliminate pointers and represent data in a *dense*, serialized form, such as the Gibbon compiler. We propose a new memory management strategy for language runtimes with mostly serialized heaps. It uses a hybrid, generational collector, where regions are bump-allocated into the young generation and objects are bump-allocated within those regions. Minor collections copy data into larger regions in the old generation, compacting it further. The old generation uses region-level reference counting. The resulting system maintains high performance for data traversal programs, while significantly improving performance on other kinds of allocation patterns.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Garbage collection**.

Keywords: Compilers, Garbage Collection, Data Representation

https://doi.org/10.1145/3652024.3665512

1 Introduction

Since manual memory management is extremely error-prone, high-level languages have, for several decades, employed automatic memory management (AMM), such as tracing garbage collectors and reference counting. During this time, the assumptions about the target language runtime varied just like the design of languages themselves, and the particulars of an AMM system is often influenced by the language it is attached to. Attempts to design generic memory systems have appeared over time but they still reflect assumptions from their originating languages and implementations, and new language designs can prompt new, unforeseen challenges for AMM.

Most established AMM designs target language runtimes where program values are represented with pointers to small objects allocated sparsely on the heap. These runtime objects are atomic in two senses: first, they never change their shape, and second, there are no sub-parts of the objects that are individually addressable—the targets of pointers elsewhere in the heap. Another common assumption is that an object has the size of at least one pointer.

Recently, a novel approach to representing program values as dense structures (i.e. pointer-free, serialized byte arrays) has been studied. This has huge performance advantages [11, 15, 18] due to minimized scattered memory accesses and maximized data locality. Several works have explored computing directly with serialized data representations of varying density: Cap'N Proto [27], FlatBuffers [12], Compact Normal Form (CNF) [30], Dargent [7]. Such efficiency is also achieved by the Gibbon compiler [29] for a polymorphic, higher-order, strict subset of Haskell. Gibbon compiles programs to C code and employs whole-program compilation and monomorphization.

In Gibbon's implementation, the heap hosts dense structures residing in *regions*, which represent growable units of allocation. Each region contains a single *value* (corresponding to a logical value from the source language), which is made up of numerous *objects* (allocations corresponding to data constructors); e.g., a list value can consist of several cons-cell-objects. Objects in the same region are packed side-by-side and pointers between them are the exception rather than ubiquitous[1].

[1]Consider, for example, the following definition of the `List` datatype: `data List = Cons Int List | Nil`. The `Cons` data constructor has the `Int` value inlined immediately after a one-byte constructor tag; the tail is treated the same and inlined immediately after the 8-byte integer. So, a list with 100 elements can fit in 901 contiguous bytes: 9×100 plus one byte for the `Nil` constructor.

With densely represented values, pointers have to be employed sometimes to recover some of the flexibility or space efficiency (sharing) of the traditional pointer-based approach. E.g. in Gibbon, so-called *indirection pointers* preserve asymptotic space and time complexity, e.g., when compiling a program that shares common data between two values. The compiler automatically extends the datatype to include a *tagged indirection pointer*, e.g. `Ind (Ptr List)`, as an implicit extra case in the sum type. Crucially, such pointers allow objects to refer to objects in other regions and make Gibbon heaps only *mostly* serialized.

Prior work on AMM for mostly serialized heaps used region-based memory management (MLkit [23], UrWeb [9]). In these works, object lifetime is determined automatically by the compiler and it depends on the lexically-scoped region the object is assigned to. Object lifetimes must be conservative and may be overly long, in the worst case, equivalent to the entire program execution. In pathological cases, this can lead to unbounded space leaks [23]. Gibbon's approach is similar, but indirection pointers can cause objects to stay alive beyond their region's lexical scope, thereby prompting the need for an additional region-level reference-counting scheme [14]. In contrast, traditional tracing garbage collectors, while not garbage free, can bound the heap size using the semi-space strategy [8]. For that reason, MLKit later added backup garbage collection *within* regions [10].

Tracing collectors are attractive but cannot be used with mostly serialized heaps directly. Supporting denser data representations violates some of the usual assumptions baked into many memory management systems, namely: (1) the heap consists of objects of statically-known sizes (plus arrays) connected by pointers, (2) pointers always refer to the starting address of the target object, and (3) every data constructor (e.g. a node in a tree or a cons-cell in a list) is its own independent object in memory at runtime.

This paper tackles the above challenges and presents an AMM system for mostly serialized heaps in the context of the Gibbon project. Our contributions are as follows:

- The first practical and complete solution to automatic memory management in mostly serialized heaps. We adopt a generational collector design, with copying collection for the young generation while keeping Gibbon's reference-counted regions for the old generation. This hybrid approach (similar to Ulterior [3]) enables fast bump-allocation of new objects and regions, while retaining efficient handling of large and growing regions.
- Novel design choices prompted by our unusual setting. We introduce new regions during collection time, to keep all roots valid after collection (Section 3.1). Because values continue growing after promotion to the old generation, we allow allocation into old-generation regions (Section 3.3), which, in turn, incurs a need for a remembered set. We allow sharing, in spite of the fact that many objects are smaller than a word-sized pointer, and thus develop a new approach to forwarding pointers (Section 3.4).
- An evaluation of the resulting system—GC-Gibbon—on both tree-traversals, which are favorable for the serialized data approach, and benchmarks that stress small object allocation, where Gibbon used to be at its weakest (Section 5.1). We show that GC-Gibbon retains Legacy-Gibbon's strong performance on tree-traversals and attains reasonable performance on out-of-order, small allocations, approaching mature systems. For small allocations, our system is 3.79×, 0.46×, and 1.09× geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For tree-traversals, our geomean speedup is 1.02×, 2.19× and 1.5×. Gibbon's memory consumption reduces to 47% (geomean) on the small allocation benchmarks. We also evaluate our design choices in Section 5.2.

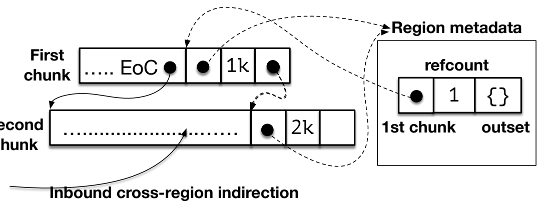


Figure 1. Run time representation of regions in Gibbon (and GC-Gibbon's old generation). This particular logical region is made up of two chunks, with an end-of-chunk pointer linking the data. There's an incoming tagged indirection which causes the refcount to be 1. Pointers occuring in the data are shown with solid arrows, whereas other implementation related pointers are shown with dashed arrows.

2 Anatomy of Mostly Serialized Heaps

In this section we overview mostly serialized heaps as implemented in Gibbon and give examples when the approach shines or falls short.

To serialize the data representation Gibbon uses LoCal [14, 28] (**Lo**cation **Cal**culus), a first-order intermediate representation (IR) with an explicit byte-addressed, mostly serialized data layout. Gibbon converts input programs into LoCal programs using *location inference* (a variant of region inference [23, 25]), and thereby makes explicit the byte-level data layout of all values.

Regions, objects, and chunks. In LoCal, all values reside within *regions*. Every region itself resembles a heap with a single allocation cursor, which is used to perform all writes in it. The allocation cursor always points to the next available cell on the heap, and new objects are bump allocated. *Locations* are addresses within a region that are used to read and write raw data. LoCal always allocates objects in a region in order—every location in a region must be written to before a new writeable location can be created after it. This serial ordering imposed on locations is what serializes the resulting value.

There are two subcomponents to each object: there is the *fixed portion* of an object, consisting of a tag, and any constant-sized fields such as integer scalars. Then there is the *extended portion* of the object which fills a variable number of bytes due to child objects being "inlined" within its representation. Even the extended object may be smaller than the complete logical *value*, which would include all memory reachable by the transitive closure of any indirections.

Gibbon allocates a constant-sized *chunk* of contiguous memory for each fresh region, as Figure 1 shows. When this chunk is exhausted, a new one which is double in size is allocated and linked with the previous one using a pointer. This doubling policy is used up to an upper bound, after which constant-sized new chunks are allocated[2].

In order to detect if a chunk is exhausted, every write operation needs to know where the current chunk ends, so that it can perform bounds checking. For this reason, every location is dilated to be a pair of `(alloc,end)` cursors at run time. To mark that the serialized stream of data continues in another chunk Gibbon implicitly adds yet another reserved constructor—and thus one-byte tag value—to each datatype: `(EoC Ptr)`, which signals an end of chunk and stores a pointer to the head of the next chunk. We refer to these pointers as *end-of-chunk pointers*. When a reader hits an `EoC` tag, they must use the stored pointer to resume reading. We refer to read or write pointers into chunks as *cursors* because of their largely contiguous motion.

Sharing via tagged indirection pointers. Tagged indirection pointers enable a program to share a value among multiple locations. For example, one might write code to construct a binary tree node as

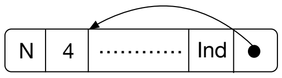
```
let x = buildTree n in Node n x x
```

and expect a single, shared value to be allocated for the left and right subtrees. But without a way to store the addresses of previously allocated values in the data representation, the right subtree would have to be allocated by copying the entire left subtree at the appropriate location. To avoid copying, Gibbon implicitly compiles every datatype `d` to have an additional reserved constructor `(Ind (Ptr d))`, which stores an absolute pointer to a value of type `d`. Given such additional constructors, Gibbon can compile the above code to the following:

```
let x = buildTree n in Node n x (Ind (addrOf x))
```

In this version, Gibbon can ensure that the call to `buildTree` allocates directly in place, in the left subtree after the `Node` tag. Thus a pointer is needed only for the right subtree:

[2]Keeping the initial chunk small is optimal in situations where a region contains a small value. But if a region needs to grow to store a large value, the doubling policy would amortize the overall allocation overhead.



```
data List = Cons Int List | Nil

mkList 0 = Nil
mkList n = Cons n (mkList (n-1))
```

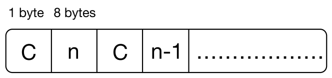


Figure 2. Gibbon source code for `mkList` and in-memory representation of its result. `C` is short for `Cons`.

```
reverse Nil acc = acc
reverse (x:xs) acc = reverse xs (x:acc)
```

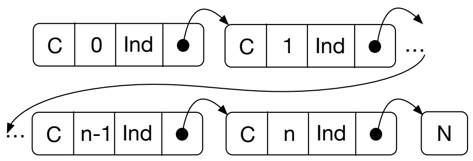


Figure 3. Gibbon source code for accumulator-style list reversion and in-memory representation of its result. `C` is short for `Cons` and `N` is short for `Nil`.

Indirection pointers are critical to Gibbon's ability to compile functions without changing their asymptotic complexity. They provide "opt in" pointers, with the default case being dense serialization, and the exceptional case being indirection. The runtime overhead (branching on a one byte `Ind` constructor) is placed on the exceptional case.

The Splendors and Miseries of Gibbon. Dense representations of data are sensitive to allocation patterns of algorithms in use. Here we show one example where Gibbon is at its best—*in-order* allocations—and another one where Gibbon's approach suffers the most—out-of-order allocations.

Figure 2 shows the Haskell source for a simple `mkList` function, as well as the resulting structure in memory. Thanks to the destination-passing style [21] LoCal employs, Gibbon is able to produce densely represented list of integers in the first chunk of the output region. If the list does not fit in one chunk, Gibbon will allocate more and connect the two together. This representation is still far more efficient than the traditional linked structure where every node involves an isolated allocation.

Gibbon programs use bump allocation in regions to achieve efficient, linear allocation patterns, such as that of `mkList`, where the result value, the constructed list, is laid out in a single region. But if a function's allocation pattern differs, such as when reversing a linked-list or when building a binary tree by allocating its right subtree before the left, the resulting value is placed across multiple regions. Consider the Haskell code for the standard accumulator-style list-reverse function given in Figure 3. For every successive recursive call, Gibbon has to allocate a new region for the new accumulator value, since it does not have any other region to use (the destination region for the result cannot be used yet because the intermediate values of the accumulator do not have an obvious connection to the overall result). As a result, every output cons cell goes in a separate region, linked together using indirection pointers—a traditional linked list!

Programs like `reverse` which allocate small regions at a very high rate show the overheads of region allocation and collection in Gibbon. In fact, the list-reverse program compiled by Gibbon is 4× slower than its pointer-based version. This isn't surprising because the Gibbon program performs the same number of region-`malloc`'s as the pointer-based version does for objects, but also does additional work to track region metadata information such as the reference counts and outsets (pointers to regions to which objects in this region point, shown in Figure 1).

Another problem that is highlighted in Figure 3 is that of fragmentation. First, reverting to a pointer-based representation means all subsequent traversals of this list would be slower because of poor data locality and pointer chasing. A little slower than in a traditional pointer-based heap, because of processing the extra tag-check on each indirection pointer. Second, the space usage is not efficient, with only one `Cons` cell per region, most space is left unoccupied.

3 Design of GC-Gibbon

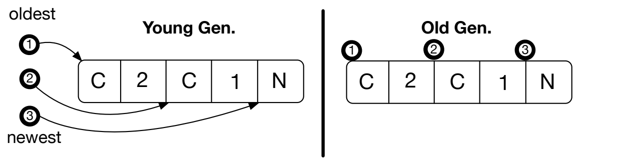
GC-Gibbon adopts a copying generational collector with compacting ability for its young generation and keeps Gibbon's reference-counted regions for the older generation. The old regions, however, use deferred reference counts now.

All new chunks are allocated in the young generation by bumping the allocation pointer. When the young generation becomes full (i.e. exceeds a certain threshold), it has to be collected: live data is copied to the old generation and the young generation is reset. We retain Gibbon's reference-counted regions for the old generation. After copying is complete, any old-generation regions that are dead as per the deferred reference counting mechanism are freed. In the future we plan to extend this with an additional tracing collector to collect regions in the old generation.

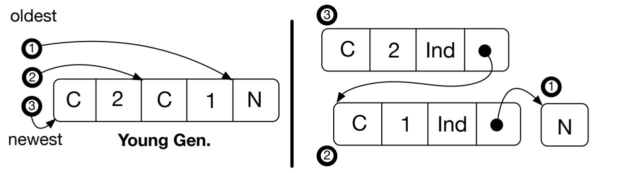
In the next sections, we explain how we treat a *minor collection*: tracing the young generation starting from the roots and copying to a different region representation in the old generation where regions are granted their own growing, memory allocations, and equipped with extra metadata.

3.1 Managing GC Roots

Shadow Stack. We use a *shadow-stack* [13] to maintain a root set of live objects for collection. Following the convention, addresses of objects (locations in LoCal) that are live after an allocation point are spilled to the shadow-stack, and are restored later, potentially having been updated by an intervening collection. Spilling is also necessary across function call sites, as the function might perform allocations.



(a) Evacuating GC roots from oldest to newest creates a compact old-generation object with no indirection pointers.



(b) Evacuating GC roots from oldest to newest creates an old-generation object with unnecessary indirections.

Figure 4. Rootset sorting. The root numbers are indicative of their evacuation order; root 1 is evacuated before root 2, and so on.

Along with the spilled location, we also spill and restore the end-of-chunk address for the chunk that the spilled location belongs to. There are two reasons: (1) if the location is in a young-generation chunk, it will be promoted to the old generation, and its end address will change, and (2) if it's in an old-generation chunk, the collector would require its end address to access the region-level metadata (reference count, outset etc.) whose address is stored at the footer. Also, the type of the value residing at the spilled location is stored on the shadow-stack. It serves as a numeric index into an *info-table* (described in Section 4) that stores layout information required to guide the evacuation routine.

Evacuation order of GC roots. Like other tracing collectors, the order in which roots are processed influences the layout and locality of the resulting old-generation heap [1]. However, our collector is even more sensitive to this ordering: certain traversal orders produce a compact heap with no indirection pointers, while some others produce significantly pointer-heavy heaps.

To illustrate sensitivity of our collector to the root ordering, we consider the young generation and the root set in Figures 4a and 4b. The young-generation object is identical in both cases, but the order of GC roots corresponding to the child objects is different. In Figure 4a, the oldest root corresponds the parent object, and younger roots correspond to child objects deeper in the graph. In Figure 4b, the roots are ordered in the opposite order. This ordering of roots in the root set is a consequence of different workloads: in-order allocators like `mkList` tend to create root sets like Figure 4a, while reverse-order allocators like `reverse` tend to create root sets like Figure 4b. If a collection is triggered at this point, there is no fixed order of root set traversal (e.g. newest-to-oldest versus oldest-to-newest) that would handle both examples efficiently[3]. For these list examples, an efficient traversal starts evacuation with the head of the list, promoting the entire list into a single old-generation region (Figure 4a). An inefficient traversal would evacuate the tail of the list first, and then subsequently evacuate earlier parts of the list which would be linked by (unnecessary) indirections to the already-evacuated portions (Figure 4b).

The key insight here is to pick an ordering that consistently evacuates upstream data earlier, irrespective of the order of allocation. Specifically, we want a traversal order that (1) evacuates roots for the newer regions before the older regions, and (2) within each region, evacuates roots that are towards the beginning of the region before roots that are towards the end.

We get a favorable ordering as follows. First, we bump the allocation pointer of the young generation backwards. That is, the allocation pointer starts at the end of the young generation (the high address), and moves towards the start (the low address). Next, we sort the root set such that roots corresponding to objects at lower addresses appear before those at higher higher addresses. This policy gives us both the desired properties because objects towards the beginning of the region already occupy addresses lower than those of objects towards the end, and the reversed bump allocation puts newer regions at addresses lower than those of older regions.

For a desirable post-collection heap, indirections should only be proportional to actual sharing in the data. To quantify this notion more precisely, we define an optimal heap with minimum indirection count:

Definition 3.1 (Minimal indirections post-collection)**.** The minimum number of indirections post-collection includes: up to one per root, plus $N - 1$ indirections for every object which has $N > 1$ references to it.

If a live object has only a single reference to it, then it must be placed in the same region as the object from which it is reachable. The per-root indirections are there because the roots themselves represent pointers into the heap data. Further, we place these pointers in fresh regions only if the object has not been copied already, otherwise, the root is directly updated to point to the object's new address.

Our post-collection heaps always achieve minimal, sharing-only indirections, as in Definition 3.1. This compactness comes at the cost of sorting the roots. In practice, root sets are small enough that the time spent on sorting is worth the resulting improvement. Also, the root set is bounded by the program stack size. If we wanted to further bound sorting time to a constant, we could use a partial sorting algorithm, or, completely skip sorting, trading it off against having more indirections in the post-collection heap.

[3]We say that a traversal order is efficient if it produces compact heaps and introduces minimum unnecessary fragmentation.

3.2 Bird's-Eye View of Evacuation

We now sketch the algorithm to evacuate each root in the root set. Here we focus on evacuating completely written objects, deferring the discussion of partially-written objects to Section 3.3 and sharing to Section 3.4

The collector processes all roots present on the shadow-stack one by one. For each root, it first checks whether a young-generation object has already been evacuated, in which case it updates the root to have the the corresponding addresses of the relocated data in the old generation. Otherwise, it allocates a fresh region in the old generation to relocate this object. This fresh region can grow during collection, using the normal policy of doubling the size of each additional chunk in the linked series of chunks. Then, the collector learns the kind of object it is evacuating by inspecting the first tag in the object, based on which it carries out the evacuation. Once evacuation is complete, the start and end addresses of the root are updated. The pseudo-code of the evacuation algorithm is given in Appendix D.

The first tag of an object defines this object's kind and informs evacuation. Below we explain the strategy for every object kind.

Tagged indirection pointer: If the target of the pointer is in the young generation, it is inlined by copying its data. Otherwise, the indirection pointer is copied as it is. In the latter case, the indirection pointer written by the collector is an old-to-old pointer, and the reference count of the region containing the target object must be updated as our (deferred) reference counts include only old-generation pointers.

Updating reference counts requires accessing the target region's metadata, which must be findable given only the contents of the tagged indirection pointer being copied. To accomplish this we use the 64-bit indirection pointers to pack in both (1) the address of the target object, and (2) the offset from there to the target chunk's footer—which is where the region metadata resides. We give details of this pointer encoding in Section 4.

Recall that the mutator uses indirection pointers to share objects among different regions. There is additional work required here to carry forward this sharing into the old generation. We postpone this discussion to Section 3.4.

End-of-chunk pointer: Due to the pretenuring policy (Section 3.3) the target of an end-of-chunk pointer is always in the old generation, and thus an object is considered completely evacuated upon reaching this tag. But, we still need to combine metadata information for two regions: (1) the fresh region that was created to copy the object that ends in this end-of-chunk pointer, and (2) the old-generation region that was created earlier due to pretenuring to store the remainder of this object. For this reason, all end-of-chunk pointers also need to encode an offset from their target to the target chunk's footer.

Burned or forwarded object: Object that has already been evacuated, discussed in Section 3.4.

Cauterized object: Partially-written object, discussed in Section 3.3.

Regular data constructor: If the tag is not among the reserved tags listed above, it corresponds to an allocation of a regular data constructor and is copied by referring to the info-table (Section 4). Its fixed portion, consisting of a tag and constant-sized fields, is copied directly using `memcpy`. The extended portion, consisting of child objects occupying variable number of bytes, is processed by recursively[4] inlining the children into the destination region.

3.3 Growing Partially-Written Objects

Certain objects encountered by GC-Gibbon's copying collector might be only partially written. For example, the mutator could be in the middle of allocating a tree structure when it triggers a collection, which leaves the young generation with a region containing a tree node having a left field but no right field (yet). When such a tree node is promoted to the old generation, the collector must stop copying after the left field, otherwise it will keep reading uninitialized data. Thus the collector must be able to detect the ends of such partially-written objects. Furthermore, once the minor collection is complete the collector must decide where to grow this object, that is, where to restart construction of the remainder of the object (the right field)—in the young or in the old generation.

We use a region's allocation cursor (Section 2) to detect partially-written objects. Every region has a single allocation cursor, where the next object would be written. As a corollary, if a region does not have an allocation cursor, it cannot have any partially-written objects in it. Before beginning copying, the collector writes a special reserved tag at all live allocation cursors, effectively *cauterizing* the regions to mark the end of initialized data. The copying routine described in Section 3.1 stops copying upon reaching reaching this tag, so as to not read any uninitialized data. To support this, the mutator spills all live writeable locations to a separate shadow-stack before starting a collection, and restores their updated addresses after the collection is complete. Regions that contain a fully constructed value do not have an allocation cursor, as they do not have any writeable locations in them. Correspondingly, such regions do not undergo cauterization and the live objects within them are promoted in the standard way.

[4]We use a worklist instead of call-stack based recursion for efficiency.

The design choice of where to restart construction of the remainder of partially-written values is a tradeoff between (1) requiring a write barrier for new indirection pointers written into the old generation, and (2) sacrificing the benefits of pretunuring large and growing regions into the old generation.

Design choice 1: Restarting construction in the young generation: We would allocate one young-generation chunk for each partially-written object that was promoted to the old generation, and, to continue its construction, update its allocation cursor to point to the beginning of the new chunk. Next, we would use an end-of-chunk pointer to link the end of the promoted object to the beginning of this fresh young-generation chunk. These end-of-chunk pointers (pointing from the old generation to the young generation) would serve as a *remembered set* of roots for the subsequent minor collection. However, crucially, all writes would now always happen in the young generation. As a result, and since LoCal is a pure language, this remembered set would remain constant until the subsequent collection. Moreover, all new indirection pointers would be young-to-old pointers, and could be created without a write barrier (for maintaining a separate remembered set), making them fairly cheap to create. With this policy, we would use a remembered set of end-of-chunk pointers that is updated once per collection, instead of a remembered set of indirection pointers that is maintained by the mutator using a write barrier.

While this policy reduces the cost of indirection pointers, it precludes the collector from performing pretenuring [5, 19, 26], which requires that the mutator be able to allocate certain long-lived objects directly in the old generation. This would have a significant impact on the performance of programs that allocate large values, often using a small number of regions. Such programs would exhaust the young generation frequently and trigger a collection. Moreover, programs that have a large number of regions under construction simultaneously could cause an exceptional situation where after the minor collection, a large portion of the young generation is populated by these new chunks for older promoted values, also increasing the number of collections (unless the young generation is allowed to grow).

Design choice 2: Restarting construction in the old generation: In GC-Gibbon we use the dual of the previous choice, namely to continue the construction of partially-written objects in the old generation. After an object is promoted, its allocation cursor is updated to point to the frontier of its old-generation chunk, and no new chunks are created in the young generation. As discussed above, the choice to allow allocations directly in the old generation has two consequences: the collector can perform pretenuring, but pointer creation needs to be protected by a write barrier since the remembered set of indirections pointing from the old to young generation can dynamically grow[5]. While such a write barrier is expensive, we already amortize its overhead by minimizing the number of indirection pointers. Also, this write barrier is no worse than what already exists in Gibbon, which has to potentially update reference counts and outsets when creating indirection pointers. On the plus side, pretenuring is vastly beneficial for programs that allocate large data structures—exactly the kind of bulk-data-processing programs which are Gibbon's speciality.

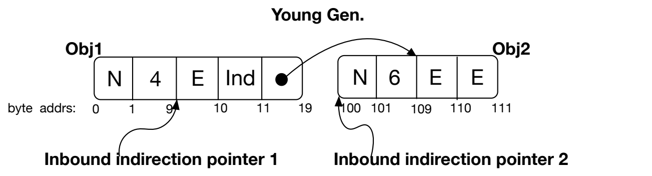
We adopt the following pretenuring policy: the first chunk of every region is allocated in the young generation, but all subsequent chunks directly start in the old generation. After a small prefix, the remainder of a large structure would be written only once and never copied by the collector, similar to Gibbon. The reasoning behind this policy is that the lifetime of a structure is at least the time required to construct it, which could be quite large for large structures. Analogous to the accepted wisdom that old objects tend to live even longer, large regions are more likely to grow even bigger. Also, in our experience typical Gibbon programs tend to not require many indirection pointers, so we try to optimize the more common case. This design choice has a big impact on benchmarks evaluated in Section 5.1 (Table 2).

3.4 Maintaining Sharing During Collection

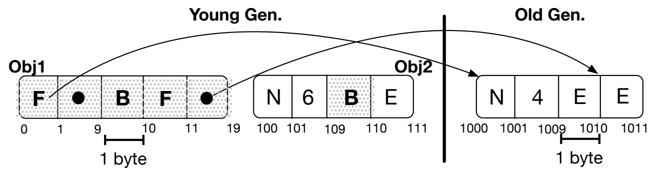
An object can have more than one inbound pointer, and thus a tracing collector must maintain sharing as it relocates data. For example, the rootset can contain stack variables with multiple pointers into different parts of the same object, as is the case when the mutator recurs through multiple levels of a tree. If we fail to detect sharing while copying live data, all the local variables on the stack may end up with their own copy of the data, which is wasteful.

There are several challenges to maintaining sharing given a dense, mostly serialized heap. First, there is insufficient space for forwarding pointers inside many objects' layouts. Second, when the collector is mid-way through copying a object, and finds a sub-portion of it has already been copied, it needs a *skip-over address* to skip over the already-copied portion and resume copying after it. The simplest solution to both these challenges is to maintain two *side-metadata* tables during collection: (1) to store all forwarding information, i.e. a table that maps (start,end) intervals in the young generation to their corresponding addresses in the old generation, and (2) another table to store skip-over addresses. Unfortunately, using these tables naively is expensive and makes the collector several times slower. Instead, we explore an approach that stores this metadata in the copied portion of objects where possible, and only uses side-metadata tables as a fallback (the slow path), as we explain next.

[5]The remembered set can never shrink, because pointers in the old generation cannot be deleted by the mutator as LoCal is a pure language.



(a) A young-generation heap with two objects, `Obj1` and `Obj2`. `Obj1` has one inbound indirection pointer from an object not pictured here. `Obj2` has two inbound indirection pointers: from `Obj1` and from another object not pictured here. `N` is short for `Node` and `E` is short for `Empty` (a binary search tree constructor).



(b) The young-generation heap from (a) after `Obj1` has been evacuated. The collector has been able to add two forwarding pointers, but one object (`E`) has been burned without a forwarding pointer.

Figure 5. An in-progress evacuation that illustrates how objects are *forwarded* and *burned*.

Forwarding: The forwarding strategy we use follows the principle of: (1) precisely marking each byte that is copied, while (2) opportunistically including forwarding pointers anywhere in the bytestream where there is room: including wherever indirection pointers exist, and any data-constructors with more than a pointer-sized quantity of scalar data. We say that data marked in this manner is either forwarded, by writing a forwarding pointer into its payload, or, when too small for forwarding, *burned*, with each status corresponding to another reserved tag value. We denote these tags `B` and `F`. A "`B`" behaves like a single byte object, whereas a 9 byte "`F addr`" object consists of the forwarded tag followed by the new address of the object which previously lived at the same byte location as the forwarded tag. This is illustrated in Figure 5a and Figure 5b.

When the collector needs to compute the forwarded address of a given tag, it either reads it directly (if the object was forwardable), or it reads a burned tag and scans to the right looking for a forwarding entry within the same span. Once such an entry is found, the address of the original tag byte in question can also be computed via subtraction—as its location relative to the forwarded object in the from-space and to-space will be conserved. This conservation holds because: (1) inlining of data due to the presence of indirection pointers is the only reason why an object could have different sizes in the from-space and the to-space, and (2) each indirection itself has enough space to store a forwarding pointer after being evacuated. Thus, the forwarding address of any object occurring in the span of bytes serialized before an indirection pointer can be computed in a straightforward manner: we use the forwarding pointer that will be written in place of the old indirection.

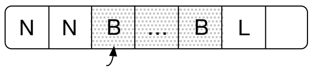
Consider a scenario where the from-space heap is as illustrated in Figure 5b and the collector follows “inbound indirection pointer 1” and reaches the burned tag in `Obj1` at byte-address `9`. It will now scan to the right and immediately discover a forwarding pointer at byte-address `10`. This forwarding pointer points to the to-space byte-address `1010`. The collector will compute the forwarded address of the object at byte-address `9` as: `1010 - (10 - 9) = 1009`.

Our goal in designing the sharing-preservation aspect of our collection algorithm is to bound the amount of scanning time necessary to resolve a forwarded address for any tag residing in the from-space before collection. Nevertheless, for completeness we need to also introduce a global table as the place of last resort to store forwarding information. The table maps (start,end) intervals in the from-space to their corresponding start locations in the to-space. The collector may enter in the middle of a burned interval, so the table needs to be an interval map allowing forwarding-address lookups keyed on locations anywhere within the forwarded interval.

We fall back to the table when we fail to find forwarding information by scanning. We can fail by hitting the end of the span, which we recognize based on encountering an already-burned tag of the next span. The collector also exposes a *max_scan* parameter for the maximum number of burned bytes that should be traversed in search of a forwarding pointer. By default this is set to the young-generation chunk size. After scanning this many bytes, we fall back to the global table to lookup forwarding information.

Symmetrically, with writing as with reading, after reaching the end of the span or burning *max_scan* bytes, without successfully forwarding, we populate an entry in the table. Because the bound must hold from any starting location, after successfully forwarding an object, we begin counting again, up to *max_scan*. If we hit an indirection, it is itself a forwardable object. In this case, we also mark the object downstream from it as start of a new span (in addition to resetting the *max_scan* counter).

Skipping-over: A worst-case scenario is, for instance, a tree value with shape only, containing no scalars, but whose subtree was already copied and burned. This value consists exclusively of small, non-forwardable objects:



Scanning to the end of this interval will fail to turn up a forwarding pointer. Fortunately, the size of the problematic interval is bounded by the chunk size of the starting region in the young generation. The reason is that, otherwise, the interval would contain a chunk redirection pointer to another chunk, which itself is forwardable. Unfortunately, the landlocked, evacuated value needs to not only have its new forwarding address resolved (for writing an indirection in the to-space), but the collector also needs to know where the burned value ends in the from space, so that it can subsequently continue collection.

To this end, our algorithm introduces a second table for storing skip-over addresses. The table provides a fallback and a slow path for evacuation, just like the table for forwarding pointers. However, our collector creates a table entry only when it enters a region for the first time at a non-zero location[6]. The reason is that values which begin at location zero in a region never need to be skipped over in this way. Either they are top-level values, or if they are referred to, it is via an indirection, which itself is trivial to skip over. Furthermore, such location-zero values are always forwardable, because we leave enough room in the footer of each region chunk to store a forwarding pointer. Conversely, when the collector stumbles on a transition to burned data in the middle of a chunk, it switches to the slow-path, performing a table lookup.

To support skipping ahead, the table stores only the end of the entire burned interval, i.e. the end of the value rooted in the first burned byte. This information is sufficient because the collector can only jump into the middle of the burned interval by following an indirection there. In that case, forwarding information is needed but skip-over information is not, because it is trivial to skip-over the value by skipping over the indirection itself. Adjacent burned intervals are not ambiguous with a single interval, because they correspond to two logical values, and as such are copied by separate evacuations resulting in separate entries in the table. In Figure 5b, our collector would create an entry in the skip-over table for the object at byte-address `109`, since this object lives at a non-zero location and is burned by following an indirection pointer. The collector will look up this entry in the table if it evacuates the object starting at byte-address `100`.

In conclusion, this subsection introduced a sharing preserving strategy, which, for completeness, includes side-metadata tables for slow-path lookups of forwarding and skip-over information. However, we expect that programs take these slow paths rarely, because: (1) heaps generally have a sufficient density of primitive data (ints, floats, strings) or indirections such that there is a high density of forwardable objects, and (2) skipping over already-evacuated values is necessary only when the nursery contains sharing. The latter case generally happens in programs that perform fine-grained allocations of small regions, with a high percentage of objects occupying location zero of their respective regions.

[6]In future work, static analysis may assist in ruling out sharing and lessening this obligation.

3.5 Write Barrier

As explained in Section 3.3, because we allow old-to-young pointers, we need a write barrier on indirections written in the old generation. On every write of an indirection, we test the target address to see if it's in the nursery, and if not we add it to a remembered set. Our write barrier currently compares the pointer against a $(start, end)$ range of addresses for the nursery. Further optimizations to the "is in nursery?" predicate are possible in the future[7].

One important optimization we perform is to prevent redundant chains of indirections, *short circuiting* them, in the write-barrier. We perform an additional load to peek at the tag of the target to which the new indirection points, and if it is an indirection, we keep following it until we find a non-indirection tag as the target. As with other aspects of the design, this leverages the immutability of our heap to maintain the invariant.

4 Implementation Details

We implement our AMM system in the open-source Gibbon compiler[8]. Our implementation mainly changes Gibbon's runtime system with only a few additions to certain LoCal-to-LoCal compiler passes. In the runtime, the region creation routine is updated to create a chunk in the young generation using bump-pointer allocation.

Our garbage collector is implemented in the Rust programming language primarily because of its performance characteristics along with a rich the standard library. The Rust code is compiled as a dynamic system library (using `crate-type=["cdylib"]`) and then linked with C code generated by GC-Gibbon. This choice has a side effect: we lose potential compile-time or link-time optimization opportunities between the C and Rust code[9]. However, we limit the interaction between our C and Rust code to just one function call, `garbage_collect`, which reduces any potential slowdowns caused due to missed optimizations.

Info-table: We use a statically allocated info-table to store the layout information required to evacuate objects of different types. This table is populated by the program when it starts executing. For each user-defined datatype in the source program, the info-table has an entry of type `DatatypeInfo` (given in Figure 12 in the Appendix). The main evacuation loop operates like an interpreter consuming a stream of byte-codes; when the object being evacuated starts with a tag corresponding to a regular data constructor, (as opposed to a reserved tags desribed earlier) it retrieves the necessary layout information from the info-table.

[7]For example, we could allocate the nursery in a fixed portion of the virtual address space, so that a test on the pointer is sufficient for the is-in-nursery test, without any additional loads for (dynamic) nursery bounds.

[8]https://github.com/iu-parfunc/gibbon/

[9]Since we use GCC to compile the generated C programs because it usually produces more efficient code than Clang in our experience, especially for the switch-heavy tree traversal programs. Besides, getting meaningful link-time optimizations between code compiled using Clang and Rust is not trivial.

Pointer encoding: At various points during collection, the collector needs to know metadata information of a region which houses an object that is the target of a pointer (indirection pointer or end-of-chunk pointer). For instance, if writing an old-to-old indirection pointer, the target region's reference count needs to change; if promoting a chunk that ends with a link to a pretenured chunk, the target region's set of chunks need to be updated. These metadata can be accessed via the footer of the target chunk. To get to the footer, we use a 64-bit pointer to store both: the address of the target object and the offset from there to the target chunk's footer. The offset is stored in the 16 high-order bits. As a result, the maximum chunk size in our system is bound at 65K bytes (2^{16}).

Optimizing default region size: A performance anti-pattern with previous versions of Gibbon was to allocate a sizeable region of the default size, typically at least 1K, and then write only a single constant-sized object to it, such as one cell of a linked list. This wastes a lot of memory, and in GC-Gibbon, can also cause many more collections to occur. It is therefore profitable to identify certain regions with *statically bounded maximum size*. We add such a static analysis on Gibbon's LoCal intermediate language. When the compiler backend generates code for region allocation, it overrides the default size with the static bound if it is smaller. Implementing this requires analyzing all the locations that allocate to a particular region, and then inferring the sizes of objects written to these locations. The size of primitive types such as ints and floats is known a priori. Expressions that allocate a variably-sized serialized value (for example, using recursion) are inferred to have an *unbounded* size.

5 Evaluation

In this section we evaluate our memory management system using a variety of benchmarks taken from previous literature, and two additional benchmarks—`reverse` and `treeUpdate`—that stress the worse-case scenarios for our implementation. Besides prior Gibbon (referred to as Legacy-Gibbon in the rest of this section), we compare the performance of our implementation to GHC[10], which is especially optimized to run functional programs which allocate lots of small objects, and Java, which has a highly optimized and mature garbage collector. For our experiments, we use a single-socket Intel E5-2699 18 core machine with 64GB of memory and running Ubuntu 18.04. We compile the C programs generated by our implementation using GCC 7.5.0 with all optimizations enabled (option `-O3`). For comparing against Legacy-Gibbon, we use its version `0.2` compiled from source. To ensure an

[10]https://www.haskell.org/ghc/

Table 1. Run times of out-of-order and small-allocation benchmarks (in seconds and relative to GC-Gibbon).

	GC-Gibbon	Legacy-Gibbon		GHC		Java	
Benchmark	T_{gcg}	T_{lg}	$T_{\text{lg}}/T_{\text{gcg}}$	T_{ghc}	$T_{\text{ghc}}/T_{\text{gcg}}$	T_{java}	$T_{\text{java}}/T_{\text{gcg}}$
reverse	0.49	1.46	2.98	0.42	0.86	0.53	1.08
treeUpdate	0.77	4.17	5.41	0.37	0.48	0.56	0.73
coins	4.34	35.5	8.18	1.21	0.28	3.63	0.84
lcss	0.51	0.30	0.59	0.45	0.88	0.72	1.41
power	1.40	8.07	5.76	0.28	0.20	2.36	1.68
geomean	-	-	3.39×	-	0.46×	-	1.09×

Table 2. Run times of in-order allocation and bulk-traversal benchmarks (in seconds and relative to GC-Gibbon).

	GC-Gibbon	Legacy-Gibbon		GHC		Java	
Benchmark	T_{gcg}	T_{lg}	$T_{\text{lg}}/T_{\text{gcg}}$	T_{ghc}	$T_{\text{ghc}}/T_{\text{gcg}}$	T_{java}	$T_{\text{java}}/T_{\text{gcg}}$
buildKdTree	2.67	2.53	0.95	7.78	2.91	4.48	1.68
countCorr	1.77	1.77	1.00	3.00	1.7	4.47	2.52
allNearest	0.71	0.80	1.13	1.46	2.06	1.00	1.41
barnesHut	3.54	3.40	0.96	5.83	1.65	2.40	0.68
constFold	1.38	1.50	1.09	4.12	2.98	2.56	1.85
geomean	-	-	1.02×	-	2.19×	-	1.5×

apples-to-apples comparison, we port our bounding-region-size optimization (Section 4) to Legacy-Gibbon. For GHC, we use GHC 9.0.2, with options `-threaded -O2`. We use GHC's default collector [16] and control the size of its young generation with the run-time option `+RTS -A <SIZE> -RTS`. For Java, we use OpenJDK 17.0.1 with its default collector and control the size of its young generation with the option `-XX:NewSize=<SIZE>`. Each reported measurement is the mean of 10 runs, where each run records the wall-clock time required to run a benchmark. For Java, we do two additional runs to warm up the JVM but don't count their run time when computing the mean. We oberved low variance in all our measurements and therefore do not report it separately.

Benchmarks: We use two sets of benchmark programs:

(1) programs in Table 1 perform many out-of-order and small-allocations where the mostly serialized approach is weak,
(2) programs in Table 2 allocate or traverse a large data structure, where the mostly serialized approach shines.

(We give brief descriptions of all benchmark programs in Appendix C.1.) For GHC, we use strict datatypes in benchmarks, which generally offers the same or better performance, but avoids performance complications due to laziness. All programs use the same algorithms[11] and datatypes, and are run with the same inputs. For GHC and Legacy-Gibbon, we hold the size of the young generation constant at 4MB. For Java, the young generation starts with a size of 4MB, but is allowed to grow if desired by the collector.

[11]To workaround a stackoverflow error, we use `for` loops instead of recursion for the Java implementation of `reverse` and `treeUpdate`.

Table 3. M is the total memory allocated by Legacy-Gibbon. For GC-Gibbon, N_{coll} is the number of minor collections, M_Y is the memory allocated in the young generation across all collections, M_O is the memory allocated in the old generation, and R is the relative memory usage calculated as, $(M_Y + M_O)/M * 100$.

	Legacy-Gibbon	GC-Gibbon			
Benchmark	M	N_{coll}	M_Y	M_O	R (%)
reverse	1.6 GB	64	256 MB	146 MB	25
treeUpdate	5.9 GB	612	2.6 GB	5 MB	44
coins	44 GB	1815	7.2 GB	757 MB	18
lcss	337 MB	18	75 MB	425 MB	148
power	17 GB	3083	12.9 GB	54 MB	76
geomean	-	-	-	-	47

5.1 Evaluating Run Time and Memory Consumption

Run Time Performance. Tables 1 and 2 show the run time results of GC-Gibbon to Legacy-Gibbon, GHC, and Java on simple tree traversals from either of two classes: out-of-order or in-order.

For small out-of-order allocation benchmarks (Table 1), GC-Gibbon benefits from its fast bump-allocated young generation, whereas Legacy-Gibbon shows the overheads of `malloc`-based region allocations. In the case of `reverse`, both Gibbon versions need to allocate a new region per input element, and thus 8M regions are allocated in this instance. But despite this very high rate region allocation, GC-Gibbon is 7% faster than Java and only 14% slower than GHC. The `lcss` benchmark is surprisingly fast with Legacy-Gibbon. According to our initial observations, `lcss`' allocation pattern seems to naturally have a stack-like behavior and benefits from Legacy-Gibbon's region based memory management.

For in-order allocation and bulk-traversal benchmarks (Table 2), Legacy-Gibbon has a home-turf advantage. Importantly, this advantage is not harmed under GC-Gibbon. Key to this resilience is the pretenuring optimization described in Section 3.3—for these benchmarks, only the first chunk is allocated in the young generation and the rest are directly allocated in the old generation. Due to this, Legacy-Gibbon and GC-Gibbon also have similar memory usage for these benchmarks. The slowdowns observed here are primarily because GC-Gibbon's pointer encoding mechanism, which puts an upper bound on the largest chunk that it can allocate, namely 65K, unlike Legacy-Gibbon which sets this upper bound to 1GB. However, both GC-Gibbon and Legacy-Gibbon outperform GHC and Java on these benchmarks.

For small allocations, our system is 3.79×, 0.46×, and 1.09× geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For bulk-tree-traversals, our geomean speedup is

Table 4. Run times in seconds of benchmarks run with different GC configurations (explained in Section 5.2).

Benchmark	Default	NoBurn	NoCompact	Simple-Barrier	NoBurn +SB	NoCompact +SB
reverse	0.49	0.43	11.8	0.49	0.44	11.9
treeUpdate	0.77	0.75	0.88	2.30	1.20	12.1
coins	4.34	4.32	4.31	10.45	10.3	10.4
lcss	0.51	0.53	0.54	0.52	0.52	0.53
power	1.40	1.34	1.36	1.38	1.40	1.44

1.02×, 2.19× and 1.5×. Overall, these results show that GC-Gibbon offers significant performance improvements compared to Legacy-Gibbon on small and out-of-order allocation benchmarks, without degrading the performance on bulk-traversal and allocation benchmarks. Legacy-Gibbon is extremely slow on certain workloads such as `coins`, `power` [17] or `treeUpdate`, thereby discounting its use entirely if any part of an application has allocation patterns like these.

Memory Consumption. Table 3 shows the memory allocation behavior for the benchmarks from the small-allocation group. GC-Gibbon has geomean 47% memory usage relative to Legacy-Gibbon. Aligning with the runtime results where `lcss` was GC-Gibbon's weakest spot, the memory consumption of this benchmark is higher compared to Legacy-Gibbon.

Sensitivity to Size of Inputs. Figure 6 shows the run times of benchmarks using inputs of various sizes. The young-generation size is held constant at 4MB. All the variants have similar behavior, with Legacy-Gibbon being the slowest in most cases. The graphs for `reverse` and `treeUpdate` show how GC-Gibbon fixes especially poor asymptotics of Legacy-Gibbon.

For memory consumption, we observe close to constant improvement (as measured by the *R* column in Table 3) when varying the input size.

Other parameter sweep results are posted in Appendix C.2.

5.2 Evaluating Our Design Choices

To evaluate the effects of the design choices we made, we run the benchmarks that stress the collector in six different modes, each of which toggles a specific choice. All of these modes (except `SimpleBarrier`) are configuration flags provided to the collector. `SimpleBarrier` requires recompiling the mutator since the write-barrier is inlined into the mutator at compile time.

- **Default:** follow the design described in Section 3, with all optimizations enabled.
- **NoBurn:** disable the forwarding pointer mechanism (Section 3.4). and allow shared values to multiply. A benefit is that writing forwarding pointers, burning data, and maintaing side-metadata tables is not needed.
- **NoCompact:** disable compaction (pointer elimination). For each indirection pointer encountered during evacuation, if the target object is not already copied, it is put into a fresh region and a new indirection pointing to this fresh region is created.
- **SimpleBarrier:** disable elimination of redundant chains of indirections (Section 3.5). This makes the write-barrier more efficient by reducing the number of memory loads it performs, but makes the collection more expensive because of the overheads associated with evacuating indirections (due to forwarding, side-metadata tables, etc.).
- **NoBurn+SB**: combine `NoBurn` and `SimpleBarrier`.
- **NoCompact+SB**: combine `NoCompact` and `Simple-Barrier`.

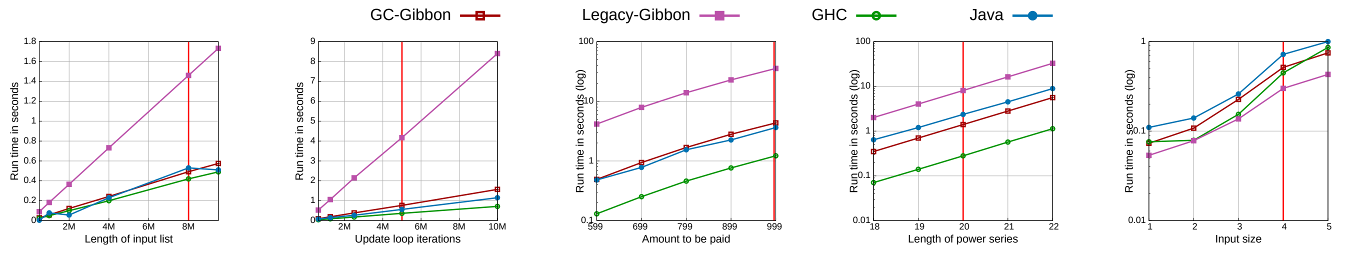
The results are given in Table 4. At first glance, `NoBurn` mode looks like the best choice. It is fast because it touches less memory in the young generation, as it does not have to write forwarding pointers or maintain side-metadata tables. But, since sharing is disabled, it is very easy to run into a pathological worst-case that could cause the collector to copy an exponential amount of data, leading to inefficient space usage. None of the benchmarks we considered here trigger this behavior, however. With respect to `NoCompact` mode, `reverse` is 24× times slower in this configuration since it has the highest number of indirections among these benchmarks—8M, one for each cons-cell. The effectiveness of the indirection-chain-eliminating write-barrier is demonstrated by `treeUpdate` and `coins`, both of which create long indirection chains and are 2-3× slower in this mode.

6 Related Work

The most closely related work to this paper is, of course, Vollmer et al.'s Gibbon compiler [29], which was summarized in Section 2. This paper presents a generational collector with copying collection for the young generation while keeping Gibbon's reference-counted regions for the old generation.

Region-based Memory Management: The main motivation behind *region-based memory management*[2, 22, 25] was to bring some of the benefits of *stack-based memory management* (a technique common in imperative languages like Pascal and Algol) to higher-order functional languages, primarily Standard ML. In this context, "region types" are a feature of a type system that tracks what region of memory a value is allocated into, with the goal of safely deallocating all values in that region once it goes out of scope. This approach suffers from lack of prompt deallocation of memory (Section 1). Also, some common patterns of functional programming (e.g. tail recursive functions) can end up causing memory leaks [23].

Various attempts at extensions or optimizations to region systems have been proposed to address this, such as *storage mode analysis*, where a compiler inserts special instructions to reset the allocation pointer in a region if it can probe the region contains no live values, essentially allowing its



(a) reverse **(b)** treeUpdate **(c)** coins **(d)** power **(e)** lcss

Figure 6. Run times in seconds of benchmarks using inputs of various sizes, the young-generation size is held constant at 4MB. The red line marks the input used for measurements reported in Tables 1 and 3.

contents to be overwritten [24]. Elsman and Hallenberg [10] explored combining regions and copying collection in the MLKit system. While MLKit uses regions, each object within a region is traditional in every other way; the heap is *not* serialized like it is in Gibbon. Its collector does not face the challenges of copying serialized, partially-written objects, but also doesn't benefit from the resulting compaction.

Garbage Collection: The literature on traditional garbage collection includes not only much work on tracing and reference counting independently, but also in combination. Our proposed collector is similar to Ulterior reference counting [3] which also has a copying young generation and a reference-counting old generation. This work was further extended in [20], which uses an efficient heap structure [4] and reference counting, and also includes a backup tracing collector. More recent work [31] has shown the performance benefits of using an efficient heap structure and reference counting. The LXR collector brings together several optimizations and heuristics, and introduces an efficient remembered set and a low-overhead write barrier to make reference counting efficient, and is able to reclaim most memory without any copying. There is ample opportunity to improve our reference counting collector using these techniques.

Compressed heap data: Memory compression is available at the page level as an operating system feature, but it has also been explored in the language runtime, for example in Java for embedded devices [6]. More recent work on computing with serialized data [27], can be viewed as a form of compression.

7 Conclusions and Future Work

We presented a new approach to memory management for mostly serialized heaps, as found in the Gibbon compiler and its runtime system. This hybrid collector is able allocate objects and regions quickly and coalesce objects, which were scattered at the points of their allocation, into efficient, serialized representations.

This work is the first step in a new direction that invites further study and refinement. It is common in computer science to trade-off time and space using compression techniques, and these mostly serialized heaps point to opportunities to explore these tradeoffs more deeply in the context of language's in-memory representations. More prosaic, there are additional optimizations to develop and apply to our system to further close the gap with traditional implementation techniques on their "home turf", (i.e. the worst-case scenarios for Gibbon's native representations). We also plan to extend the reference counting strategy in the old generation with an additional tracing collector. Finally, a major topic of future work is to scale the approach to the parallel setting, both for the mutator and the collector.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and comments. This work was supported in part by National Science Foundation awards CCF-1908504, CCF-1919197, CCF-2216978, CCF-1909862, CCF-2119352, as well as by the Engineering and Physical Sciences Research Council award EP/X021173/1.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An Efficient Parallel Heap Compaction Algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 224–236. https://doi.org/10.1145/1028976.1028995

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/237721.237771

[3] Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA) *(OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/949305.949336

[4] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/1375581.1375586

[5] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[6] Guangyu Chen, M Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, Bernd Mathiske, and Mario Wolczko. 2003. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. 282–301.

[7] Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. 2023. Dargent: A Silver Bullet for Verified Data Layout Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 47 (jan 2023), 27 pages. https://doi.org/10.1145/3571240

[8] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (nov 1970), 677–678. https://doi.org/10.1145/362790.362798

[9] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

[10] Martin Elsman and Niels Hallenberg. 2020. On the effects of integrating region-based memory management and generational garbage collection in ML. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 95–112.

[11] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing) (SC '13)*.

[12] Google. 2014. FlatBuffers. https://google.github.io/flatbuffers/

[13] Fergus Henderson. 2002. Accurate Garbage Collection in an Uncooperative Environment. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) *(ISMM '02)*. Association for Computing Machinery, New York, NY, USA, 150–156. https://doi.org/10.1145/512429.512449

[14] Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. *Proc. ACM Program. Lang.* 5, ICFP (2021).

[15] Junichiro Makino. 1990. Vectorization of a treecode. *J. Comput. Phys.* 87 (March 1990), 148–160. https://doi.org/10.1016/0021-9991(90)90231-O

[16] 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 International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1375634.1375637

[17] M. Douglas McIlroy. 1999. Power Series, Power Serious. *J. Funct. Program.* 9, 3 (may 1999), 325–337. https://doi.org/10.1017/S0956796899003299

[18] Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. 2011. Data Parallel Programming for Irregular Tree Computations, In HotPAR. https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/

[19] Andreas Sewe, Dingwen Yuan, Jan Sinschek, and Mira Mezini. 2010. Headroom-based pretenuring: dynamically pretenuring objects that live "long enough". In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* (Vienna, Austria) *(PPPJ '10)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/1852761.1852767

[20] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the Gloves with Reference Counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 93–110. https://doi.org/10.1145/2509136.2509527

[21] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/3122948.3122949

[22] Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894

[23] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[24] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value Lambda-Calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 188–201. https://doi.org/10.1145/174675.177855

[25] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[26] David Ungar and Frank Jackson. 1988. Tenuring policies for generation-based storage reclamation. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (San Diego, California, USA) *(OOPSLA '88)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/62083.62085

[27] Kenton Varda. 2015. Cap'n Proto. https://capnproto.org/

[28] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

[29] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

[30] Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. 2015. Efficient Communication and Collection with Compact Normal Forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 362–374. https://doi.org/10.1145/2784731.2784735

[31] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-Latency, High-Throughput Garbage Collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 76–91. https://doi.org/10.1145/3519939.3523440

A Appendix: Sample Gibbon programs

In this section we present the details of some Gibbon programs, their translated LoCal IR code, and finally the C code that is generated by Gibbon.

A.1 mkList: constructing a linked list

```haskell
data List = Cons Int List | Nil

mkList :: Int → List
mkList 0 = Nil
mkList n = Cons n (mkList (n-1))
```

(a) `mkList`'s starting Haskell source code.

```
mkList :: ∀ l^r. Int → List@l^r
mkList n = if n == 0 then Nil l^r else
    letloc l_1^r = l^r + 9 in
    let tail: List@l_1^r = mkList [l_1^r] (n-1) in
    Cons l^r n tail
```

(b) `mkList` compiled into LoCal IR by Gibbon

Figure 7

```c
typedef struct GibCursorProd3_struct {
    GibCursor field0; GibCursor field1;  GibCursor field2;
} GibCursorProd3;

GibCursorProd3 mkList(GibCursor footer_r_289, // Output region's footer.
                      GibCursor loc_288,      // Output cursor.
                      GibInt n_30_97_141      // Length of the linked list to create.
) {
    // Check if the output region has more space, grow the region otherwise.
    if ((loc_288 + 18) > footer_r_289) {
        gib_grow_region(&loc_288, &footer_r_289);
    }
    if (n_30_97_141 == 0) {
        *(GibPackedTag *) loc_288 = 1;            // Nil
        GibCursor after_tag_573 = loc_288 + 1;
        return (GibCursorProd3) {footer_r_289, loc_288, after_tag_573};
    } else {
        GibCursor loc_335 = loc_288 + 9;
        *(GibPackedTag *) loc_288 = 0;            // Cons
        GibCursor after_tag_582 = loc_288 + 1;
        *(GibInt *) after_tag_582 = n_30_97_141;  // Element in the cons cell
        GibCursorProd3 tmp_struct_5 = mkList(footer_r_289, loc_335, (n_30_97_141 - 1));
        return (GibCursorProd3) {tmp_struct_5.field0, tmp_struct_5.field1, tmp_struct_5.field2};
    }
}
```

Figure 8. `mkList` translated to C by Gibbon.

This code operates as follows. The arguments of `mkList` include the number `n` and a symbolic location l^r, where the resulting list of length `n` will be allocated. This allocation technique is a form of destination-passing style.In LoCal, a location l^r is an address within in a region r and type $\tau@l^r$ is assigned to a value of type τ residing at a location l^r. If `n` is zero, `mkList` simply allocates `Nil` at location l^r. Otherwise, it starts constructing a cons cell. First it binds a location $l_1{}^r$ that is 9 bytes past the location l^r (one byte for the `Cons` tag and 8 bytes for the integer). Next, it recursively constructs a list of length `(n-1)` at location $l_1{}^r$. Then it writes the `Cons` tag and the integer `n` starting at location l^r, which completes the construction of this cons cell. Gibbon's location inference algorithm places the cons cell and its tail within the same region because the allocations in `mkList` happen *in order*. Thus, the resulting list looks like the structure shown in Figure 2.

A.2 reverse: reversing a linked list

```
reverse :: List → List
reverse Nil acc = acc
reverse (x:xs) acc = reverse xs (x:acc)
```

(a) Starting Haskell source code for accumulator-style list-reverse.

```
reverse :: ∀ l^r m^s n^t. List@l^r → List@m^s → List@n^t
reverse xs acc = case xs of
  Nil → Ind n^t (addrOf acc)
  Cons (y:Int@l_y^r) (ys:List@l_ys^r) →
    letregion u in
    letloc o_1^u = start u in
    letloc o_2^u = o_1^u + 9 in
    let cpy: List@o_2^u = Ind o_2^u (addrOf acc)
    let acc': List@o_1^u = Cons o_1^u y cpy in
    reverse [l_ys^r o_1^u n^t] ys acc'
```

(b) reverse compiled into LoCal IR by Gibbon.

```
typedef struct GibCursorProd6_struct { GibCursor field0; GibCursor field1; GibCursor field2; ··· } GibCursorProd6;

GibCursorProd6 reverse(GibCursor end_r_293, GibCursor end_r_294, GibCursor end_r_295,
                       GibCursor loc_292, GibCursor xs_31_98_145, GibCursor acc_32_99_146)
{
    GibShadowstack *rstack = READ_SHADOWSTACK;
    GibShadowstack *wstack = WRITE_SHADOWSTACK;
    GibShadowstackFrame *frame;
    if ((loc_292 + 18) > end_r_295) {
        gib_grow_region(&loc_292, &end_r_295);
    }
    GibPackedTag tmpval_931 = *(GibPackedTag *) xs_31_98_145;
    GibCursor tmpcur_932 = xs_31_98_145 + 1;
    switch (tmpval_931) {
      case 0: {
            GibInt tmpval_937 = *(GibInt *) tmpcur_932;
            GibCursor tmpcur_938 = tmpcur_932 + sizeof(GibInt);
            gib_shadowstack_push(rstack, acc_32_99_146, end_r_294, Stk, PList_T);
            gib_shadowstack_push(rstack, tmpcur_938, end_r_293, Stk, PList_T);
            gib_shadowstack_push(wstack, loc_292, end_r_295, Stk, PList_T);
            GibChunk region_939 = gib_alloc_region(32);
            GibCursor r_356 = region_939.start;
            GibCursor end_r_356 = region_939.end;
            frame = gib_shadowstack_pop(wstack);
            loc_292 = frame→ptr;
            end_r_295 = frame→endptr;
            frame = gib_shadowstack_pop(rstack);
            tmpcur_938 = frame→ptr;
            end_r_293 = frame→endptr;
            frame = gib_shadowstack_pop(rstack);
            acc_32_99_146 = frame→ptr;
            end_r_294 = frame→endptr;
            GibCursor loc_345 = r_356 + 9;
            *(GibPackedTag *) r_356 = 0;
            GibCursor after_tag_600 = r_356 + 1;
            *(GibInt *) after_tag_600 = tmpval_937;
            gib_indirection_barrier(loc_345, end_r_356, acc_32_99_146, end_r_294, PList_T);
            return reverse(end_r_293, end_r_356, end_r_295, loc_292,  tmpcur_938, r_356);
        }
      case 1: {
            GibCursor jump_412 = xs_31_98_145 + 1;
            gib_indirection_barrier(loc_292, end_r_295, acc_32_99_146, end_r_294, PList_T);
            GibCursor end_592 = loc_292 + 9;
            return (GibCursorProd6) {end_r_293, end_r_294, end_r_295, jump_412, loc_292, end_592};
            break;
        }
      case GIB_INDIRECTION_TAG: {
            uintptr_t tagged_tmpcur_17 = *(uintptr_t *) tmpcur_932;
            GibCursor tmpcur_956 = GIB_UNTAG(tagged_tmpcur_17);
            uint16_t tmptag_958 = GIB_GET_TAG(tagged_tmpcur_17);
            GibCursor end_from_tagged_indr_447 = tmpcur_956 + tmptag_958;
            GibCursor jump_449 = tmpcur_932 + 8;
            GibCursorProd6 tmp_struct_16 = reverse(end_from_tagged_indr_447,  end_r_294,
                                              end_r_295, loc_292, tmpcur_956, acc_32_99_146);
            return (GibCursorProd6) {end_r_293, tmp_struct_16.field1, tmp_struct_16.field2,
                                 jump_449, tmp_struct_16.field4, tmp_struct_16.field5};
        }

      case GIB_REDIRECTION_TAG: {
            uintptr_t tagged_tmpcur_19 = *(uintptr_t *) tmpcur_932;
            GibCursor tmpcur_971 = GIB_UNTAG(tagged_tmpcur_19);
            uint16_t tmptag_973 = GIB_GET_TAG(tagged_tmpcur_19);
            GibCursor end_from_tagged_indr_447 = tmpcur_971 + tmptag_973;
            return reverse(end_from_tagged_indr_447,  end_r_294,
                           end_r_295, loc_292,  tmpcur_971, acc_32_99_146);
        }

      default: {
            printf("%s\n", "Unknown tag in: tmpval_931");
            exit(1);
        }
    }
}
```

Figure 10. reverse translated to C.

A.3 treeUpdate

```
loop :: ∀ $l^r$ $k^s$.
  RNG → Int → Tree@$l^r$ → Tree@$k^s$
loop [$l^r$ $k^s$] rng i tr =
  if i == 0
  then Ind $k^s$ (addrOf tr)
  else
    let (n,rng') = next rng 0 512 in
    letregion t in
    letloc $j^t$ = start t in
    let tr':Tree@$j^t$ =
      if n % 2 == 0
      then insert [$l^r$ $j^t$] n tr
      else delete [$l^r$ $j^t$] (n-1) tr
    in loop rng' (i-1) tr'
```

```
insert :: ∀ $l_1{}^r$ $k_1{}^s$.
  Int → Tree@$l_1{}^r$ → Tree@$k_1{}^s$
insert [$l_1{}^r$ $k_1{}^s$] n tr = case tr of
  Null → Leaf $k_1{}^s$ n
  Node (m:Int@$l_m{}^r$) (x:Tree@$l_x{}^r$) (y:Tree@$l_y{}^r$)→
    if m < n then
      letloc $k_2{}^s$ = $k_1{}^s$ + 1 in
      let x':Tree@$k_2{}^s$ =
        Ind $k_2{}^s$ (addrOf x) in
      letloc $k_3{}^s$ = after Tree@$k_2{}^s$ in
      let y':Tree@$k_3{}^s$ =
        insert [$l_y{}^r$ $k_3{}^s$] n y in
      Node $k_1{}^s$ m x' y'
    else ... -- Insert into the left subtree.
  Leaf (m:Int@$l_m{}^r$) → ...
```

Figure 11. A LoCal program that repeatedly inserts a random number into a binary search tree.

B Appendix: Implementation Details

B.1 Reordering Tag Allocations

Even though allocations in LoCal happen in order, its formalism requires that a data constructor tag be written *after* all its fields are. However, this creates a problem for our collector, which might need to copy a value while it is still under-construction (Section 3.3). Without the tag present at the beginning of an object, the collector cannot infer what kind of an object it is copying. We bypass this by reordering the writes such that a data constructor tag always gets written before any of its fields. Not only does this help the collector, it also makes the mutator slightly more efficient.

B.2 Random-access via shortcut pointers

While indirection pointers enable allocation of values out-of-order, they do not enable reading values out-of-order. To this end, Gibbon uses untagged shortcut pointers to enable constant-time random-access to certain fields on a per-data-constructor, per-field basis. The reason shortcut pointers are necessary is that some programs need to "skip over" certain parts of a value to read it out-of-order, and there is no way to accomplish this if the value is fully serialized—the only way to access a particular part in it is to scan past all of the data that has been serialized before it. For example, to compile a program which fetches the rightmost leaf of a binary tree with the correct asymptotic complexiy ($O(logn)$), Gibbon stores the absolute address of the right subtree in each intermediate node 7 so that it can be accesssed directly without traversing the left subtree, which would make this a $O(n)$ operation. Figure 4 shows such a node.

Fortunately, shortcut pointers are innocuous with respect to garbage collection because they always point within the same region, and thus cannot change the lifetime of other regions. In principle, they require less space to store than in a normal pointer-based representation, because no pointer is needed for the leftmost non-scalar field, e.g. one pointer for a binary tree, instead of two. A possible implementation choice would be to store the integer size in bytes of packed fields which can be used to skip over them. For example, the address of the right subtree can be computed as (the address of the left subtree + size of the left subtree), given that all the fields are serialized side-by-side in a single region. However, the runtime representation of regions (chunked, growable, as described in the next section) makes this choice slightly less efficient in practice.

B.3 Info-table representation

Figure 12 gives the representation of the info-table used in GC-Gibbon's runtime system.

C Appendix: Evaluation

C.1 Benchmark descriptions

- **reverse**: This is the standard accumulator style list-reverse program shown in Figure 3; it reverses a list containing 8M integers. The Java implementation is defined using a `while` loop rather than a recursive function.

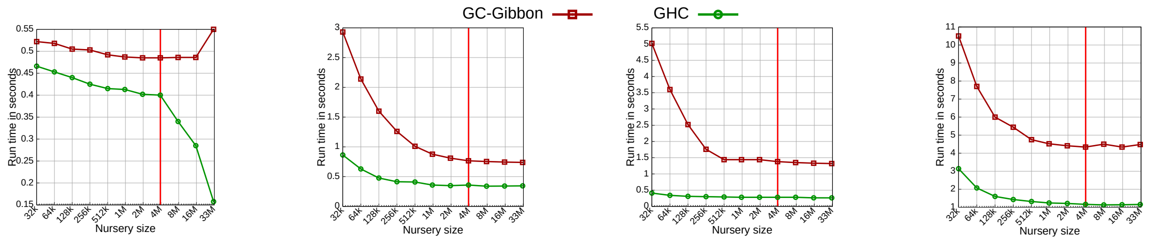
```
type InfoTable = Vec<DatatypeInfo>;

enum DatatypeInfo {
    Scalar(usize),
    Packed(Vec<DataconInfo>),
}
```

```
struct DataconInfo {
    /// Bytes before the first packed field.
    scalar_bytes: usize,
    /// Number of shortcut pointer fields.
    num_shortcut: usize,
    /// Field types of packed fields.
    field_tys: Vec<u32>,
}
```

Figure 12. The representation of GC-Gibbon's info-table used in runtime system.

- **treeUpdate**: This is the complete version of the program given in Figure 11 in the Appendix. It starts with a very small search tree and repeatedly inserts and deletes numbers in it. The numbers are chosen from small range (0-512) to keep the size of the tree more or less constant, and the tree is updated 5M times. The Java implementation encodes the outer "update" loop as an actual `while` loop, but `insert` and `delete` are defined using recursion in the standard way.
- **coins**: This benchmark is taken from GHC's NoFib[12] benchmark suite. It computes the number of ways in which a certain amount of money can be paid by using the given set of coins. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99), (5,122),(1,177)]`, and the amount to be paid is 999.
- **lcss**: This benchmark computes the longest-common-substring using Hirschberg's algorithm. Our implementation is taken from GHC's NoFib benchmark suite. We provide as input two strings of length 3100 and 3000 respectively, such that the result has length 2100.
- **power**: This benchmark computes 20 elements of the power series `(ts = 1 :+: ts^2)`, which is shown here assuming lazy evaluation. We use a slightly modified implementation that is suitable for a strict language.
- **buildKdTree** and **countCorr** and **allNearest**: `buildKDTree` constructs a kd-tree containing 1M 3-d points in the Plummer distribution. `countCorr` takes as input a kd-tree and counts the number of correlated (within a distance of 100 units) points for all 1M 3-d points. `allNearest` computes the nearest neighbor of all 1M 3-d points.
- **barnesHut**: Uses a quad tree to run an nbody simulation over 1M 2-d point-masses distributed uniformly within a square.
- **constFold**: This benchmark is taken from and implements constant folding for a language that supports integer arithmetic. It is run on synthetic syntax-tree which is a balanced binary tree of depth 26.

C.2 Parameter sweeps

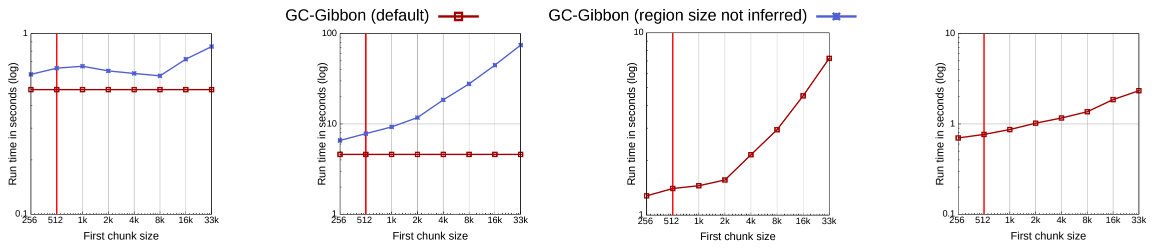


(a) reverse **(b)** treeUpdate **(c)** power **(d)** coins

Figure 13. Run times in seconds of benchmarks using young generations of various sizes. The red line marks the young-generation size used for measurements reported in Table 1 (4MB).

In this section we discuss the results of three parameter sweeps for small out-of-order allocation benchmarks that stress the collector. Figure 13 shows the run times of benchmarks using different young-generation sizes. This shows the expected tradeoff between space and time; the wall-clock time gets better as the young generation gets bigger, due to fewer collections. Figure 14 shows the run times of benchmarks using different initial chunk sizes for GC-Gibbon. Using bigger initial chunks, and therefore growing the overall region at a faster rate, causes more collections to occur since the young generation fills up faster. It also leads to inefficient space usage since many of the larger chunks will be mostly empty.

[12]https://gitlab.haskell.org/ghc/nofib



(a) reverse **(b)** coins **(c)** power **(d)** treeUpdate

Figure 14. Run times in seconds of benchmarks using initial chunks of varying sizes. Our region size inference analysis (Section 4) causes GC-Gibbon to allocate constant-sized regions in the main workoad of `reverse` and `coins`, thus their run times are constant. We report the measurements after disabling this optimization as "GC-Gibbon (region size not inferred)".

D Appendix: Evacuation Algorithm

```
-- store type layouts per data constructor (static)
global info_table[]
-- map spans of from-space memory to to-space
global fwd_table[]
-- map address of value start to its end
global skip_table[]

fun evacuate_root(from_start, ty):
  if in_oldgen?(from): -- skip root
  else if already_marked?(from):
    update_root(fwd_addr(from_start))
    else:
    (to_start, to_end) := alloc_oldgen_region()
    evacuate_object(from_start, ty, to_start)
    update_root(to_start, to_end)

-- Returns a pointer into the to-space,
-- after the given value
fun skip_over(from):
  -- We can only create a need for skipping if we process
  -- at least one data constructor in an evacuate_value:
  assert(not(zero_location?(from)))
  return skip_table[from]

-- Returns a pointer in the to-space
fun fwd_addr(from):
  -- For the first location in the region,
  -- the region metadata lets us forward:
  if zero_location?(from):
    return footer_fwd_ptr(region(from))
  else:
    while offset < max_scan:
      let next = deref(from + offset)
      match(next):
        Forwarded(addr): return (addr - offset)
        Burned: offset += 1
    -- The interval-map maps source to target bytes
    -- by internally mapping entire
    -- (src..src+k) => (dst..dst+k) ranges efficiently:
    -- side metadata lookup = slow path
    return fwd_table[from]
```

(a) Part of the evacuation algorithm. Global definitions, helpers, and entrypoint to begin evacuation. See also figure 15b.

```
-- Recursively evacuate the value at a given location,
-- with the given type. Because of the acyclic heap,
-- this will never recur back to the same location.
fun evacuate_object(from_start, ty, to_start):
  let ty_stk = [ty]
  let to = to_start
  let from = from_start
  let span_start = from_start
  let span_bytes = 0

  fun end_span():
    if span_bytes > max_span:
      let sz = from - span_start
      fwd_table[span_start .. sz] := to-sz .. to
    span_bytes := 0
    span_start = from

  -- One evacuate-and-burn session
  -- on one contiguous interval:
  while (from_ty = pop(ty_stk)):
     while (chunk_redirection?(from)):
       forward_obj(from, to)
       from := deref(from)
       end_span()
     if indirection?(from):
       -- start a separate interval in a new chunk:
       to' = evacuate_object(deref(from), from_ty, to)
       forward_obj(from,to)
       from += TAGGED_INDIRECTION_SIZE
       to := to'
       end_span()
     else if already_marked?(from):
       to := write_indirection(to, fwd_addr(from))
       if non_empty?(ty_stk):
         from := skip_over(from)
       end_span()
     else: -- regular data-copying codepath
       -- advance to,from cursors past
       -- the data written:
       (to',from') =
         write_tag_and_scalars(to, from)
       -- look up types of 0 or more
       -- non-scalar data fields (children):
       push_children(stk, info_table[from])
       fwded = burn_or_forward_obj(from)
       to   := to'
       from := from'
       if fwded: end_span()
       else: span_bytes += from' - from
    if not(zero_location?(from_start)):
      -- Populate for future slowpath lookups
      -- one byte after the end
      skip_table[from_start] := to
  return to
```

(b) Continued from figure 15a, core burning and forwarding algorithm for sharing maintenance.

Figure 15. Evacuation algorithm