Simple, Fast and Safe Manual Memory Management

Piyus Kedia
Microsoft Research, India
t-piked@microsoft.com

Manuel Costa Matthew Parkinson Kapil Vaswani Dimitrios Vytiniotis
Microsoft Research, UK
manuelc,mattpark,kapilv,dimitris@microsoft.com

Aaron Blankstein *
Princeton University, USA
ablankst@princeton.edu

Abstract

Safe programming languages are readily available, but many applications continue to be written in unsafe languages because of efficiency. As a consequence, many applications continue to have exploitable memory safety bugs. Since garbage collection is a major source of inefficiency in the implementation of safe languages, replacing it with safe manual memory management would be an important step towards solving this problem.

Previous approaches to safe manual memory management use programming models based on regions, unique pointers, borrowing of references, and ownership types. We propose a much simpler programming model that does not require any of these concepts. Starting from the design of an imperative type safe language (like Java or C#), we just add a delete operator to free memory explicitly and an exception which is thrown if the program dereferences a pointer to freed memory. We propose an efficient implementation of this programming model that guarantees type safety. Experimental results from our implementation based on the C# native compiler show that this design achieves up to 3x reduction in peak working set and run time.

CCS Concepts • **Software and its engineering** → *Allocation / deallocation strategies*; **Software safety**

Keywords memory management, type safety, managed languages, garbage collection

1. Introduction

Safe programming languages are readily available, but many applications continue to be written in unsafe languages, because the latter are more efficient. As a consequence, many applications continue to have exploitable memory safety bugs. One of the main reasons behind the higher efficiency of unsafe languages is that they typically use manual memory management, which has been shown to be more efficient than garbage collection [19, 21, 24, 33, 34, 45]. Thus, replacing garbage collection with manual memory management in safe languages would be an important step towards solving this problem. The challenge is how to implement manual memory management efficiently, without compromising safety.

Previous approaches to safe manual memory management use programming models based on regions [20, 40], unique pointers [22, 38], borrowing of references [11, 12, 42], and ownership types [10, 12, 13]. For example, Rust [5] is a recent programming language that incorporates several aspects of the Cyclone [38] design, including unique pointers, and lexically-scoped borrowing. These concepts make programming languages more complex, and often impose restrictions that require programmers to follow specific idioms. For example, unique pointers facilitate implementing safe manual memory management, but they impose severe restrictions on the programs that can be written (e.g., data structures cannot contain cycles). This often results in programmers reverting to using expensive mechanisms like reference counting (or garbage collection) or resorting to using unsafe constructs [5].

We propose a much simpler programming model for safe manual memory management that does not require regions, unique pointers, borrowing or ownership types. The starting point for our design is an imperative type safe language like Java or C#. We propose the following minimal changes to the programming model:

- We replace the garbage collected heap with a manually managed heap. Memory in this heap is allocated using the new operator, which returns a reference to a newly allocated object.
- We introduce an operator delete, which the application can use to declare that an object is no longer in use and may be reclaimed.

* A part of this work was done while the author was at Microsoft Research, UK

PLDI '17 June 18-20, 2017, Barcelona, Spain

ISBN 978-1-nnnn-nnnn-n/yy/mm. . . $15.00

DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

- We guarantee type and memory safety by introducing a new exception (DanglingReferenceException) with the following semantics: a dereference to a deleted object will either succeed as if that very object was not yet deleted, or result in a DanglingReferenceException.

The merits of this model deserve some discussion. First, this model is simple both for the compiler writer and the programmer: there are no uniqueness or ownership restrictions that affect the static type system and expressiveness of the language, no restrictions on pointer aliasing, concurrent sharing of pointers, or allocation and deallocation sites. Second, for C/C++ programmers, this programming model is familiar - it offers a similar level of control over memory usage. To get the improved efficiency of manual memory management, programmers do need to explicitly deallocate memory. We argue that this burden is well understood and acceptable to a large class of programmers, as exemplified by the C and C++ communities. Finally, unlike C/C++, we guarantee type and temporal safety. This provides stronger security since programming errors such as use-after-free bugs no longer result in memory corruptions and vulnerabilities. Memory errors result in exceptions *at the faulting deference*, and include contextual information such as a full stack trace, which make them easier to diagnose.

An important aspect of this programming model is the semantics of delete. We do not guarantee that all dereferences to deleted objects will throw an exception. While the weaker semantics is crucial for achieving good performance, it introduces non-determinism and may result in exceptions that only surface during real use. We argue that such exceptions can be detected with the combination of rigorous testing and support in the allocator for a debug mode that enforces stronger semantics (i.e. exceptions on every dereference to a deleted object) at a higher cost.

The main challenge in implementing this programming model is efficiently detecting temporal safety errors i.e. dereferences of pointers to freed memory. We propose an allocator which detects temporal safety violations using support for large address spaces in modern 64-bit processors and paging hardware. The basic method for detecting safety violations can be described as follows. The allocator assigns each object a unique virtual address; it never reuses virtual addresses (until it is safe to do so). When an object is deleted, the allocator *unmaps* the object from the application's address space. Once an object has been unmapped, the memory management unit in the processor detects any attempts to access the object and throws an access violation. The allocator catches these access violations and exposes them to the user as a DanglingReferenceException.

While the idea of ensuring temporal safety using paging hardware is promising, translating it into good end-to-end performance on modern systems is extremely challenging. In fact, several other systems [3, 4, 16, 27] have attempted to employ a similar approach. The best previous system [16] allocates each object on a new virtual page since virtual memory operations are only supported at page granularity. This results in poor performance because of high fragmentation, large number of TLB misses, and the cost of a system call on every deallocation.

Our allocator achieves good performance by allocating objects compactly (just like a conventional allocator), delaying reclamation till a page predominantly contains deleted objects, and transparently copying live objects away from page before unmapping the virtual page and reclaiming the underlying physical page. The allocator also incrementally and efficiently patches references to live objects that become invalid once the objects have been copied and the corresponding pages unmapped. The method for patching references relies on extensions to the compiler for identifying locations in the heap where a reference originates.

This approach is sufficient for a large class of applications that never exhaust the large virtual address spaces supported by 64-bit processors. For applications that come close to exhausting virtual address space, we support an incremental compaction phase that recycles virtual address space without violating safety.

Besides supporting the immediate goal of unmapping deleted objects, this design has three additional benefits. First, even if objects with different lifetimes are allocated on the same page, the copying mechanism tends to group objects with similar lifetimes; this reduces memory fragmentation which would otherwise negatively impact memory performance. Second, we can use very fast *bump* allocation – essentially just a bounds check and incrementing a pointer in the common case; the allocator does not need to worry about reducing fragmentation, because the copying mechanism handles it. Finally, and crucially, if a number of objects with similar lifetimes are allocated and deallocated in sequence (a common scenario), we just unmap the corresponding virtual pages, achieving quick physical memory reclamation without requiring any copying or compaction. In all cases, we never require tracing of the object graph through potentially very large heaps, because we rely on the programmer to specify when memory should be deallocated; this allows us to achieve large performance gains over garbage collected systems.

We have implemented the allocator in .NET native [2], a .NET runtime which supports an industrial-strength optimizing ahead-of-time compiler. Our implementation required changes to the compiler and the memory management subsystem. We have evaluated the allocator using microbenchmarks and a range of real-world applications including a key-value store, large-scale data analytics and web caching services. The evaluation shows that the allocator achieves significant improvement in performance and reduction in memory consumption (by 3X in several cases).

In summary, we make the following contributions:

- A managed runtime that replaces garbage collection with a simple programming model for manual memory management: a delete operator to free memory and an exception thrown on dereferences of pointers to freed memory.
- An allocator that guarantees type safety using a combination of paging hardware on modern 64-bit processors and a procedure for lazily patching invalid references.
- An implementation based on a production runtime and performance evaluation on large data analytics applications, as well as micro-benchmarks, showing 3X improvements in memory and CPU usage.

2. Design

2.1 Overview

As described earlier, our allocator ensures temporal safety by identifying virtual pages that predominantly contain deleted objects and copying live objects from these pages to an unused part of the address space. These pages are then unmapped and the corresponding physical pages reclaimed.

Copying objects and unmapping partially filled pages alters program execution in two ways. First, it immediately invalidates all references to live objects in registers, stack and the heap. Unlike a conventional garbage collector, our allocator does not patch these references eagerly; the application is even permitted to copy invalid references. Instead, the allocator traps any attempt to *use* an invalid reference and patches the reference to point to the new location of the object. This step relies on two features of strongly typed languages, namely the ability to walk the stack, and distinguish references from primitive values. In addition, the allocator attempts to lazily patch objects containing invalid references (with compiler support for locating containing objects). This ensures that any subsequent loads from patched objects return valid references. The allocator falls back to scanning pages and patching all invalid references only when this lazy approach generates a high rate of page faults.

Another effect of copying objects and patching references lazily is that it alters the semantics of the operation that checks equality of two references. Specifically, we can no longer assume that two references that are not bitwise equal do not reference the same object as they may refer to different copies of the same object. We therefore extend the language runtime to support an efficient algorithm for checking equality of two references, one or both of which may have been invalidated by copying. Together, both these extensions ensure that the process of copying objects and reclaiming memory is transparent to the application i.e. appears to occur as if live objects were never copied. In the rest of this section, we describe the allocator in more detail.

2.2 Heap organization

The heap managed by our allocator is organized as a set of *segments*. Each segment is a contiguous part of the address space (a power of 2, maximum of 4GB). Each segment is further divided into *generations* as shown in Figure 1a. The first generation *gen0* is half the size of the segment and each subsequent generation is half the size of its previous generation. The allocator reserves space at the start of the segment for metadata. The metadata section is organized as an array with an entry for each page in the segment. Each metadata entry contains information such as the offset of the first and last object allocated in the page and the cumulative size of allocated objects on the page.

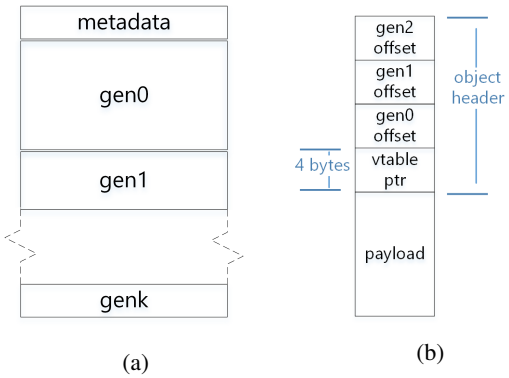


Figure 1: (a) Layout of a segment and (b) layout of a gen3 object in our allocator. The header contains the segment-relative offsets of incarnations of this object in all lower generations.

Note that our use of generations differs from generational garbage collectors. In particular, we do not intend to reuse virtual pages allocated to a generation. Instead, our allocator reclaims physical memory associated with pages in the generation that mostly contain deleted objects.

2.3 Object allocation and deallocation

Our allocator is a simple *bump allocator* with support for fast allocation from a per-thread heap. The allocator maintains a list of segments. Each application thread allocates a per-thread heap from *gen0* of the last allocated segment. It uses this heap for servicing allocation requests originating from the thread. The allocator maintains a thread-local free pointer (which initially is set to the start of the heap) and a pointer to the end of the heap. Each allocation request checks if there is enough free space in the heap, adds the object size to the free pointer and returns the previous free pointer. As long as there is space in the per-thread heap, no synchronization is required. When a thread runs out of space in its heap, it synchronizes with other threads and obtains a new per-thread heap. A new segment is allocated when *gen0* of the previously allocated segment runs out of free space.

The allocator treats objects of size greater than 32 KB as large objects. These objects are allocated from a large object heap (LOH) allocated at the bottom of the address space. In the LOH, objects do not share pages. When a large object is freed, the allocator simply unmaps the corresponding pages. Note that the allocator does not explicitly maintain the size

of each allocation. In a strongly typed language, each object stores a pointer to its type, which contains the object size.

The allocator processes delete requests by marking objects as free using a bit in the object header. The bit is also checked to detect double frees. If the deallocated object belongs to *gen0*, the allocator checks if all objects allocated on the page containing the object have been also been deleted (using per-page metadata described above). In this case, the allocator immediately unmaps the page. Any subsequent accesses to this virtual page will trigger an access violation. Pages in *gen0* that are not entirely free and pages in higher generations are reclaimed by a background process described below.

2.4 Live object promotion

As the allocator receives allocation and deallocation requests, the segment fills up with partially occupied pages, resulting in fragmentation. The allocator monitors the degree of fragmentation. When fragmentation exceeds a pre-defined threshold, it identifies virtual pages with high fragmentation, promotes live objects on those pages to higher generations and reclaims the corresponding physical pages.

Identifying candidate pages. We quantify the degree of heap fragmentation using the following measure:

$$f = \frac{\textit{total\_size\_deleted} - \textit{num\_unmapped\_pages} * \textit{PAGE\_SIZE}}{(\textit{total\_size\_allocated} - \textit{total\_size\_deleted})}$$

where *total_size_allocated* and *total_size_deleted* refer to the total bytes allocated and deleted respectively, and *num_unmapped_pages* refers to the number of unmapped pages. The numerator is the number of bytes yet to be reclaimed, and the denominator is the number of bytes allocated to live objects. When the degree of fragmentation exceeds a threshold (and enough time has elapsed since the last promotion), the allocator creates a background thread that scans the segment metadata and identifies a set of pages for promotion. The promotion policy takes into account the amount of free space on a page and the cost of promoting objects (discussed later in this section). The policy also accounts for object boundaries i.e. all pages that share a live object are promoted together. We discuss a range of policies in Section 3.

Promoting live objects. After selecting pages for promotion, the allocator enters the *promotion phase*, which executes on the background thread concurrently with mutators. The promotion phase iterates over all selected pages and promotes live objects from each page to a contiguous block of memory in the next generation.

The core of the promotion algorithm (Figure 1) is a sequence of steps that promotes live objects on a *source* page to a *target* location in the next generation. The promotion algorithm operates as follows: the allocator disables write access to the source page and then copies live objects (i.e. objects with the free bit in the header unset) to the target location. If the last live object on the source page spills over to the next page, we continue promoting objects on the next page. After copying the objects, the allocator disables write accesses to the target page(s) and updates a data structure called the *remap table*. The remap table maps each page to the offset within the segment where objects from the page have been (most recently) copied. The remap table entry for a page is updated every time objects located on that page are promoted. In other words, if live objects from page p_0 in *gen0* are promoted to an offset on page p_1 in *gen1*, and subsequently (some of) these objects are promoted to *gen2*, then the allocator updates remap table entries for both pages p_0 and p_1. Remap table entries share the space reserved for per-page allocation metadata since only one of them is required at any point in time.

Algorithm 1 Algorithm for promoting live objects on a source page to a location in the next generation.

```
1: procedure PROMOTE_PAGE(source page, target offset)
2:     make source page read-only;
3:     copy objects from source page to target offset;
4:     make target page(s) read-only;
5:     update remap table;
6:     execute memory barrier;
7:     unmap source page;
8:     enable read & write access to target page(s)
9: end procedure
```

While copying objects, the allocator also stores the segment-relative offset of the source object in the header of the target object. Since objects can be promoted multiple times through many generations, we reserve space in the object header for an offset for every generation lower than the generation of the target object (Figure 1b). The allocator uses the remap table in conjunction with per-generation offsets to locate the target object given an arbitrary source address (Section 3). After updating the remap table, the allocator unmaps the source page (reclaiming unused physical memory) and then enables write access to the target page(s).

This algorithm ensures that at any given point in time, there is at most one writable copy of each object. Also, by unmapping the source page before enabling write access to the target page, the algorithm ensures that only the copy on the target page(s) can be read thereafter.

2.5 Patching references

Unlike a conventional garbage collector, our allocator does not eagerly patch references to promoted objects. Instead, the allocator uses a *lazy* approach to patching references. The allocator installs an exception handler that detects *spurious* access violations caused by accesses to live objects on pages that have been reclaimed, and *redirects* these accesses to the corresponding promoted objects in higher generations. The allocator achieves this by identifying the origin of the

```
1  unsigned int access_violation_handler(cxt) {
2    thread *t = get_thread();
3    if (t->in_exception_context)
4      return EXCEPTION_CONTINUE_SEARCH;
5    t->in_exception_context = 1;
6    void* obj = get_faulting_address(cxt);
7  retry:
8    pobj = search_promoted_object(obj);
9    if (pobj == NULL) {
10     if (promotion_active() &&
11         is_write_exception(cxt)) {
12       sleep(1);
13       goto retry;
14     }
15     throw_dangling_reference_exception();
16   } else {
17     scan_stack_and_patch(cxt, obj, pobj);
18   }
19   t->in_exception_context = 0;
20   return EXCEPTION_CONTINUE_EXECUTION;
21 }
```

Figure 2: Access violation handler used by the allocator for handling page faults. The handler checks if the page fault is a result of dereferencing an invalid reference and patches the reference with the new location of the object.

access (i.e. register, stack or heap location that contains a reference to the object), and patching this location with the address of the promoted object. Lazy patching is suited for workloads where the number of incoming references to frequently used objects is small, and it is more efficient to patch these references when a dereference fails instead of tracing through the heap and patching all references eagerly.

In the rest of this section, we describe a mechanism for lazily patching invalid references in registers or the stack. We then describe two extensions to this mechanism for patching references in the heap without requiring heap scans.

2.5.1 Patching roots.

Figure 2 shows the lazy algorithm for patching roots i.e. references in registers or on the stack. The algorithm executes as part of an exception handler installed by our allocator. The handler receives an exception context as input (register values and type of exception i.e. read or write). The handler retrieves the faulting heap address from the context and attempts to find the address of the target object in case the object was promoted (using the routine *search_promoted_object* described in Section 3).

If the object was promoted, the handler scans all registers and the current stack frame, and patches all *stale* references to promoted objects. This step of our algorithm is similar to how garbage collectors patch roots, with a key difference that an access violation can be triggered by any load or store instruction. Therefore, we modify the compiler to emit metadata describing the location of heap references in registers and on the stack for every load and store instruction (instead of just gc safe points). Once the invalid reference(s) has been patched, execution resumes at the faulting program counter. As long as the promoted object has not been promoted again, the access will succeed. Otherwise, the access will fail and the handler will retry patching the invalid reference.

A failure to find the promoted object (line 9) indicates that the faulting object has not been promoted. However, this does not imply that the faulting reference is a dangling reference. The access violation may have occurred due to a write to the source page by another thread *before the background thread has updated the remap table*. This condition is detected using the predicate *promotion_active* which is true if the allocator is currently in the promotion phase. It is also possible that the access violation was caused by a write to the target page before write access was enabled; this is detected using the predicate *is_write_exception*. In either case, the handler busy waits for the remap table to the updated or write access to be enabled (line 13). If (and when) none of these conditions hold, then the handler generates a DanglingReferenceException (line 15).

2.5.2 Patching heap references.

Patching roots allows the mutator to recover from spurious access violations caused by promotion and make progress. However, this approach can potentially result in a large number of access violations, because registers and the stack are temporary state, and references must be patched every time they are copied from the heap into a register or stack. For example, consider the following C# code fragment.

```
int GetFirst(List<T> list) {
  Node<T> node = list.First;
  return node.Value;
}
```

If the object pointed to by the field `First` of parameter `list` is promoted, the allocator will patch the reference `node` since it is stored in a register or on the stack. However, the object `list` continues to store a stale reference to `node`. We use the term *parent* to refer to objects that contain references to a given object, and the term *child* to refer to the referenced object. We now describe two extensions for patching parent objects and thereby preventing repeated access violations.

Patching parents. The first extension is a modification to the exception handler to follow objects directly reachable from registers or the current stack frame and patch references in these objects. If no such references are not found in the current stack frame, the handler continues scanning objects reachable from the caller's stack frame. This extension is based on the heuristic that an application is likely to load and dereference objects closer to the roots. We exclude arrays from this heuristic because our experiments suggest that scanning and patching arrays eagerly can be wasteful.

As an example, consider the C# code sequence listed above. This code translates into the following instruction sequence.

```
mov r2, [r1 + 8]
mov r3, [r2 + 16]
```

Here, the register r1 contains a reference to list, the field First is located at offset 8 in the List<T> class, and Value is located at offset 16 in Node). The register r1 contains a reference to the parent of node. If the object pointed to by node has been promoted, accessing this object will fail. The extended exception handler scans the object list and patches the reference node with the address of the promoted object; any subsequent reads from list will return a reference to the promoted object.

Observe that this extension is effective only when a register or stack contains a parent when an object is accessed. In the example above, r1 happens to contain a parent when node is accessed (as an artifact of the way code is generated). However, there are several cases where this is not true. For example, if list is no longer live after the first dereference, the compiler is free to reuse r1 for storing other values. To address this, we propose two compiler transformations which ensure that when an object is accessed, a reference to the parent exists in the roots.

The first transformation extends the lifetime of variables containing references to parents to include uses of all child objects (up to the next write to the variable). If a child object is passed to a method, we extend the lifetime of the parent to include the function call. In the example described above, this transformation extends the lifetime of list to include the dereference node.Value. This ensures that list is either in a register or on the stack when the dereference occurs.

The second transformation is targeted at arrays of references. We make modifications to code generation so that operations on arrays such as array indexing explicitly expose parents in the roots. For example, consider the pattern arr[i].Value. The .NET compiler translates this pattern into the following instruction sequence.

```
mov r1, [r2 + r3*8 + 16]
mov r3, [r1 + 24]
```

Here, r2 contains a reference to the array arr, r3 contains index i and the field Value is stored at an offset 24 within element. This sequence loads a reference to the object at index i in register r1. However, there is no register or stack location containing the address of this reference. With our transformation, the compiler splits the array indexing operation into two instructions (as shown below). The first instruction loads the address of the array element into a register (r4), and the second instruction loads the reference.

```
lea r4, [r2 + r3*8 + 16]
mov r1, [r4]
mov r3, [r1 + 24]
```

This sequence requires an extra instruction and an additional register. However, it makes the reference to the parent explicit, and allows the exception handler to patch the reference in the array.

Patching dead parents. Another scenario where references to parents do not appear in the roots is when objects are returned from method calls. This is because parents in the callee stack frame disappear when the stack frame is popped. We address this scenario by instrumenting every method that returns an object to push the parent reference of the returned object into a circular buffer maintained in the runtime. We also extend the access violation handler to scan references in the buffer and patch parent objects.

The two extensions described above are not exhaustive - there are a few other cases where a reference to the parent may not exist in the roots when a child is accessed. For example, no parents exists when a freshly allocated object is accessed. Parents may also be missing if the variable holding a reference to the parent is updated before all accesses to its child objects. Our evaluation (Section 4) suggests that these cases are rare and have little impact on performance.

```
1  #define SEGMENT_OF(addr)
2    (addr & FFFFFFFF00000000)
3  #define SEGMENT_OFFSET(addr) (DWORD)(addr)
4  #define GEN_OFFSET(ref) return __lzcnt(~
       SEGMENT_OFFSET(ref))
5
6  bool reference_equal(ref1, ref2) {
7    if (ref1 == ref2) return true;
8    if (SEGMENT_OF(ref1) != SEGMENT_OF(ref2) ||
         GEN_OFFSET(ref1) == GEN_OFFSET(ref2))
         return false;
9    if (ref1 > ref2) xchg(ref1, ref2);
10   if (IS_PROMOTED(ref1)) {
11     if (!IS_PROMOTED(ref2))
12       return SEGMENT_OFFSET(ref1) == *(DWORD*)
             ((BYTE*)ref2 + OFFSET_OF(GEN_OFFSET(
             ref1)));
13     else {
14       return slow_path(ref1, ref2);
15     }
16   }
17   return false;
18 }
```

Figure 3: Pseudo-code for checking equality of two references.

Pinning references. Pinning is a primitive which prevents the memory manager for relocating an object during a specified scope. Pinning allows managed code to inter-operate with native code (e.g. system calls) by passing references to managed objects. Supporting pinning in our allocator is relatively straight-forward – we simply do not relocate pages that contain pinned objects.

2.6 Equality checks

In managed languages, the runtime is usually responsible for discharging checks for equality between references. Checking equality is an extremely frequent operation, and therefore must be fast. With our allocator, checking equality of references is challenging because references are patched lazily. Therefore, the application may attempt to check equality of two invalid references of an object that has been promoted (perhaps more than once), or compare a stale refer-

ence with a valid reference to the promoted object. In either case, a bitwise comparison of addresses in no longer valid.

Our allocator exposes a specialized operation for checking equality of two references (Figure 3). Consider the case where two references ref_1 and ref_2 are not bitwise equal. Let these references point to objects obj_1 and obj_2 respectively such that obj_2 belongs to the same or higher generation than obj_1. The checker uses the following invariants to ascertain if they are references to different copies of the same object.

- References in two different segments are not equal because objects are only promoted to generations in the same segment.
- References in the same generation are not equal.
- If obj_1 has not been promoted, then the references are not equal (since no other copy of obj_1 exists).
- The references are equal only if obj_2 has not been promoted yet and segment-relative source object offset of obj_1 is contained in the list of source offsets stored in the header of obj_2.

These checks can be discharged quite cheaply. We can compute the generation an object belongs to by counting the number of consecutive 1s in the segment-relative offset of the object. On Intel processors, this can be implemented using the `lzcnt` or `bsr` instruction (line 4). We can also check if an object has been promoted using a single lookup in the remap table (see Section 3). If these checks fails, and obj_2 has not been promoted, we use the source object offset stored in obj_2's header to determine if the references are equal. We resort to a slow path (line 14) only if obj_2 been promoted but is no longer accessible (because of page protection). In this case, we search for the location where obj_2 has promoted and check the promoted object's header. This process repeats until we find a promoted object that is accessible.

2.7 De-dangling

The patching algorithm described above incrementally fixes stale references in the heap following a promotion phase. However, in some workloads, lazily patching references can be expensive e.g. if the allocator promotes a set of live objects with a large number of active parent objects, patching each parent will incur an access violation. In such cases, the allocator falls back to the *de-dangling* phase for eagerly patching invalid references. This phase runs as a background thread concurrently with mutators. The background thread scans live objects on allocated pages. While scanning, it patches invalid references using a CAS operation. This ensures that mutator updates are not lost i.e. if a mutator updates a reference after the dedangler finds it is invalid, then the CAS will abort, or the update will overwrite the CAS.

While the de-dangling phase may appear similar to marking/tracing in a conventional GC, there is an important difference. Unlike tracing, background de-dangling is a performance optimization; it is not required for correctness. In other words, de-dangling does not have to guarantee that all invalid references are patched and can miss invalid references (such as those written to objects that have already been scanned in the current phase). This eliminates the need for a write barrier and a mechanism for tracking cross-generation pointers such as remembered sets or card tables. Furthermore, unlike tracing, which involves a traversal of the heap to find reachable objects, de-dangling is implemented as a cache-friendly, parallel scan over sets of pages since we can easily identify live objects (by checking the free bit in the header).

Algorithm 2 Pseudo-code for compaction.

```
 1: procedure COMPACTION
 2:     create space for compaction;
 3:     patch dangling references;
 4:     for all segment s do
 5:         if allocation in s is disabled then
 6:             relocate allocated objects in s;
 7:         end if
 8:     end for
 9:     patch dangling references;
10: end procedure
11:
12: procedure PATCH_DANGLING_REFERENCES
13:     dedangle heap;
14:     reset write watch;
15:     dedangle heap;
16:     suspend all threads;
17:     read write watch and patch all written pages;
18:     patch roots;
19:     resume all threads;
20: end procedure
21:
22: procedure CREATE_SPACE_FOR_COMPACTION
23:     for all segment s do
24:         disable allocation in s;
25:         try compacting objects in s;
26:         if cannot compact objects in s then
27:             enable allocation in s;
28:         end if
29:     end for
30: end procedure
```

2.8 Compaction

Recall that our allocator does not reuse virtual address space; it simply allocates a new segment when it runs out of space in the current segment. Generally, we do not expect virtual address space consumption to be a concern for most applications. However, for long running applications such as web servers and databases that allocate objects at high rates, the application will run out of virtual address space eventually. For example, an application allocating 3GB memory every

second will run out of virtual address space in a day. Our allocator supports a relatively expensive *compaction phase* (Figure 2) to handle such rare events.

The compaction phase is conceptually similar to a full heap garbage collection - it de-fragments the heap by relocating all objects in a segment to a contiguous block of memory in the beginning of *gen0* in the same segment and patching all stale references in the heap. However, *gen0* may already contain live objects and mutators actively allocating objects in previously allocated per-thread heaps. To create space for relocation, the allocator first disables allocation of new per-thread heaps in the segment, and then estimates the number of allocated bytes (k) in the segment assuming that all previously allocated per-thread heaps are fully utilized. If there is not enough free (unmapped) space in the initial k bytes of *gen0* and no mutators are actively allocating in initial k bytes, the allocator promotes all live objects in the initial k bytes to *gen1* (and higher generations if required). If there are mutator threads allocating in the initial k bytes, the allocator forces the threads to switch to other segments by suspending the threads, setting the thread-local free pointer to the end of the per-thread heap and then resuming the threads.

Once the allocator has created space in *gen0*, it can start compacting objects. However, note that at this stage, virtual pages in *gen0* may have been unmapped and must be mapped again before relocating objects. However, mapping these pages is not safe because the application may have stale references to objects originally allocated in *gen0*. Therefore, before relocating objects to *gen0*, we remove stale references to all previously allocated objects using the routine `patch_dandling_references`. This routine involves the following sequence of steps.

1. Run a de-dangling phase. This phase scans objects in the heap (concurrently with mutators), replaces references to freed objects with NULL values and patches all other references.
2. Reset write watch[1] and rerun the de-dangling phase. Write watch is an OS feature for tracking pages written by an application. Reseting the write watch clears all tracking state and allows the application to identify pages written after a certain point in time.
3. Suspend all mutator threads to prevent mutators from creating any new stale references.
4. Query and scan all pages written by mutator threads, and patch stale references.
5. Patch references in the roots of all threads.
6. Resume all mutator threads.

This routine does not guarantee that all references are patched; it only guarantees that there are no stale references to objects deleted before the routine was invoked. We run the de-dangling phase twice to minimize the amount of time spent in scanning pages while mutators are suspended. We expect the first de-dangling phase to patch a majority of stale references and the second de-dangling phase to touch only a small number of pages, which reduces the number of pages returned by the write watch.

After de-dangling all references, the allocator relocates all live objects in each segment to the beginning of *gen0* using a procedure similar to live object promotion. While relocating an object, the allocator stores the segment-relative offset of the object in the header of the relocated object in *gen0*. Once all objects have been relocated, the allocator re-runs the protocol described above for removing all stale references. Finally, we flush the remap table for the segment. The segment is now reset to a clean state (where all objects have been allocated in a contiguous block of memory at the beginning of *gen0* and there are no stale references to these objects), and can be used for allocations.

Relocating objects to *gen0* violates one of the invariants assumed by equality checking i.e. two objects that belong to the same generation in the same segment are not equal. This invariant is violated because a live object in *gen0* may be compacted to another location in *gen0*. Therefore, we extend the equality checking routine to account for relocated objects in segments where compaction is in progress. Specifically, when two references are not bitwise equal and the segment that the references belong to is being compacted, we check if one of the references has an offset smaller than the size of the compacted block of memory at the beginning of *gen0*). This indicates that the object is a relocated object. In this case, we compare the segment offset stored in its header with the segment offset of the other reference. The references are considered equal of the comparison succeeds.

3. Implementation

The allocator we propose can be integrated into any safe managed language. We have integrated this allocator with the .NET Native toolchain [2]. .NET Native is an ahead-of-time compilation toolchain for .NET applications. The toolchain consists of a compiler (based on Microsoft's C/C++ compiler), a .NET runtime and a set of libraries. The compiler frontend translates .NET binaries (in MSIL) to an intermediate language, which is then translated into architecture specific assembly code by the backend. The optimized native code is linked with the .NET runtime to generate a standalone executable. We modified the .NET runtime by disabling the GC and redirecting all allocation requests to our allocator. We also exposed a new .NET API `System.Heap.Delete` which takes an object as a parameter. We now describe a few implementation specific aspects of our design.

Searching for promoted objects. A key component of our allocator is the procedure for locating the promoted ob-

[1] `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366874(v=vs.85).aspx`

```
1  #define OBJ_TO_GEN(ref) return __lzcnt(~ref)
2  void* search_promoted_object(void* obj) {
3  retry:
4    __try {
5      if (!IS_PROMOTED(obj)) return NULL;
6
7      void* target = offset_of_first_object(obj);
8      int obj_gen = OBJ_TO_GEN(obj);
9      int target_gen = OBJ_TO_GEN(target);
10     void* cur_offset = read_offset(target,
           obj_gen);
11     void* last_offset = cur_offset;
12     int source_offset = SEGMENT_OFFSET(obj);
13
14     /* scan objects on the target page */
15     while (cur_offset <= source_offset &&
16       cur_offset >= last_offset) {
17       size = object_size(target);
18       if (source_offset >= cur_offset &&
             source_offset < cur_offset + size)
19         return target + (source_offset -
               cur_offset);
20       target += size + size_of_header(obj);
21       if (OBJ_TO_GEN(target) != target_gen)
             break;
22       last_offset = cur_offset;
23       cur_offset = read_offset(target,obj_gen);
24     }
25     return NULL;
26   }
27   __except {
28     goto retry;
29   }
30 }
```

Figure 4: Procedure for finding the address of the promoted object given a source address.

```
IS_PROMOTED(rcx):
  mov rax, 0ffffffff00000000h
  and rax, rcx  ; find segment from address
  mov r10d, ecx ; find segment offset
  shr r10d, 12  ; find page from segment offset
  mov r10d, dword ptr [rax+4*r10+
      REMAP_TABLE_START] ; load remap table entry
  test r10d, 1  ; LSB of remap table entry
      indicates if page promoted
  je ret_false
  mov rax, 1
  ret
ret_false:
  xor rax, rax
  ret
```

Figure 5: Procedure for checking if an object (in the register rcx) has been promoted.

ject given a source address. There are several ways of implementing this procedure. Our approach, shown in Figure 4), uses a remap table in conjunction with offsets stored in object headers. The procedure looks up the remap table to check if the page containing the source address has not been promoted (using the function `IS_PROMOTED`), in which case the search terminates unsuccessfully; the caller is expected to handle the condition where the page has been promoted but the remap table has not been updated yet. Figure 5 shows an implementation of `IS_PROMOTED` in machine code. This is a performance critical operation also used while checking equality of two references. Our implementation requires a single lookup in the remap table.

If a remap table entry exists, we start scanning objects from the address where the first object from the source page was copied (line 7). The scan completes successfully when we find an object header containing the source address (line 18,19). Since objects from a page are copied in order, we can terminate the search (with a failure) when we find an object that was copied from a larger offset than the source address's offset (line 15), or an object promoted from a page with a lower virtual address than the source page (line 16), or if we have scanned past the end of the generation (line 21). Objects from a page with a lower source address introduce a discontinuity in the sequence of source offsets.

Note that the search routine can trigger an access violation while scanning the target page if a different thread promotes and subsequently unmaps the target page. We handle these exceptions by restarting the search. The search will eventually succeed when the remap table entry for the source page has been updated and the new target page is not promoted during the scan.

Allocator tuning. Much like other allocators and the GC, our allocator needs to be tuned to achieve good performance over a large class of applications. There are several allocator parameters that have a large impact on performance.

The first parameter is the promotion policy, which controls when and what set of pages to consider for promotion. An aggressive policy will reclaim memory quickly but may also promote a large number of live objects, resulting in access violations. We evaluated several possible policies and eventually settled on the following: Our allocator triggers promotion when (a) the amount of memory allocated since the last promotion phase is greater than 128MB, and (b) the degree of fragmentation f is greater than 0.5. This policy is designed to keep the working set of the application to within a factor of two of the total size of live objects. When this condition holds, our allocator selects for promotion all pages where at least half the page contains deleted objects. sorts pages by degree of fragmentation and promotes pages in order until overall fragmentation f falls below $1/3$.

A second policy controls when the de-dangling phase is triggered. Although de-dangling is a background activity, it does consume CPU cycles and pollutes the caches while scanning the whole heap. However, triggering de-dangling too infrequently can cause application threads to spend a high percentage of CPU cycles handling access violations following a promotion. We control this trade-off using a single parameter: an upper bound on the percentage of CPU cycles application threads spend in handling access violations. Our experiments show that on modern hardware, the latency of each access violation is $8\mu s$, including the latency of our

access violation handler. Therefore, on an 8-core machine, an upper bound of 12.5% on CPU cycles (equivalent to 1 core) for handling access violations translates to a threshold of 125,000 access violations/sec. Our allocator triggers de-dangling once this threshold is reached. There are other policies possible e.g. policies impose per-core bounds and/or factor the number of cores the application is actively using; evaluating these policies is left for future work.

Another policy controls compaction is triggered. We trigger compaction in two rare conditions, (a) when the application is running out of virtual address space, and (b) when the application is close to exhausting physical memory.

Porting libraries. Most managed languages have a large ecosystem of libraries that support a number of high level services such as IO, strings, math, collections, synchronization, tasks etc. In .NET Native, many of these libraries are written assuming the presence of a garbage collector. We ported several libraries (such as collections, file IO etc.) to manually manage memory i.e. explicitly deleting objects using `System.Heap.Delete`. In a majority of cases, identifying the lifetime of objects (and therefore introducing `Delete`) is relatively straightforward. For example, consider the method `Resize` in the class `Array<T>`, which copies an existing array into a new array. Here, we can delete the old array right after it has been copied.

```
public static void Resize<T>(ref T[] array, int
     newSize) {
  T[] larray = array;
  if (larray.Length != newSize) {
    T[] newArray = new T[newSize];
    Copy<T>(larray, 0, newArray, 0, larray.
        Length > newSize ? newSize : larray.
        Length);
    System.Heap.Delete(larray);
    array = newArray; }
```

In other cases, we had to establish an ownership discipline and use destructors to delete owned objects (as shown below). Our allocator calls destructors just before an object is deleted.

```
public class List<T> : IList<T>, ... {
  private T[] _items;
  ...
  ~List() {
    if (_items != _empty) System.Heap.Delete(
        _items);
  } }
```

4. Evaluation

Benchmarks. Memory allocators are expected to perform well across a wide variety of applications, architectural configurations and performance metrics. Therefore, we evaluate our allocator using two sets of benchmarks. We use a set of micro-benchmarks to exercise the allocator across a wide range of loads and allocation patterns. We also use a set of diverse real-world applications namely, ASP.NET caching[2], a caching layer for web applications, RaptorDB[3], a key-value store and a simplified version of Naiad [30], a data processing engine optimized for streaming data.

Our experience in modifying these benchmarks to use manual memory management was mixed. For example, porting Naiad operators was relatively straightforward. We modified operators to explicitly free batches and internal state when they are no longer needed. We also rewrote user-defined functions (such as custom mappers and reducers) to delete any intermediate state they create. In ASP.NET caching, we added a reference count to each key-value pair to track the number of clients accessing the value. The key-value pair is deleted when the reference count falls to zero. We also extended our allocator to support notifications to indicate memory pressure. Our allocator notifies the application when the total amount of memory allocated since the last notification exceeds a threshold.

Experimental Setup. We use the .NET native toolchain to generate two (native) versions of each benchmark, one linked with the .NET garbage collector [1], and the second linked with our allocator. The .NET native GC is a background compacting collector. The GC supports two modes of operation, a workstation mode intended for client workstations and stand-alone PCs, and a server mode intended for applications that need high throughput. The workstation GC uses a single heap for all threads, whereas the server GC uses (and can collect independently) a heap per thread. Each heap contains a small object and a large object heap. The GC reserves memory for the heap in multiples of *segments*. The server GC uses comparatively larger segments than the workstation GC (4GB vs 256MB in 64-bit machines). The GC automatically adapts the size of the heap by reserving additional segments as needed and releasing segments back to the operating system when they are no longer required. For our experiments, we configured .NET native to use the server GC.

We compile all versions of the applications with the highest optimization level (-O2). We ran our experiments on a server with a quad-core, hyper-threading enabled Intel Xeon E5 1620 processor clocked at 3.60Ghz with 16GB DDR3 RAM and a 2TB SATA drive running Windows 10 enterprise. In each execution, we measure the time to completion, peak working set, and total amount of memory committed. For configurations with garbage collection enabled, we also measure the number of *gen0*, *gen1* and *gen2* collections, the percentage of time the application was paused for GC and the maximum and the average pause times. For configurations with our allocator, we measure the number of access violations and the number of times de-dangling is initiated. Each data point we report is obtained by executing the application 5 times and taking the average.

[2] `https://github.com/aspnet/Caching`

[3] `https://github.com/mgholam/RaptorDB-Document`

Benchmark	Description
GC Simulator	Data parallel benchmark used to stress test the .NET garbage collector. Each thread in the benchmark creates a collection of objects organized as a tree. The key configuration parameter is the fraction of short-lived vs. long lived objects. Threads make several passes over the collection, and randomly replace existing objects with new ones while ensuring that the fraction of short lived and long lived objects remains constant.
Red black tree	An implementation of a left-leaning red-black tree [36] with rebalancing. Each node in the tree contains a unique key and a payload of a configurable size. The tree supports lookups (by key), insertions and deletions.
RandomGraph	A graph benchmark where each vertex contains a payload and references to incoming edges. During each iteration, we pick a vertex at random, replace it with a new vertex and patch references in neighboring vertices. This benchmark represents an adversarial scenario for our allocator because each object has several parents and each parent is equally likely to be used to access the object.
ASP.NET Caching	An in-memory caching layer that can be used in ASP.NET web applications. The cache supports a get/put interface and can be configured with various expiration policies. The cache is designed to adapt to memory pressure. It registers for notifications from the GC before a *gen2* collection is about to start. A background thread clears stale entries from the cache when a notification is received.
RaptorDB	A NoSQL key-value store that supports inserts and removes of key-value pairs, where values are arbitrary bytes arrays. RaptorDB stores all values on disk and maintains an index over values in memory.
Naiad	A streaming engine that supports querying over data streams. A stream consists of a sequence of {time, record} pairs, and queries are expressed as a composition of operators such as filtering, aggregation, joins. Queries specify the *window size*, which determines the amount of time after which records expire. Operators in Naiad perform (potentially stateful) computation over one or more input batches. When a time window expires, each operator constructs an output batch and transfers it downstream.

Table 1: Benchmarks used for evaluating manual memory management.

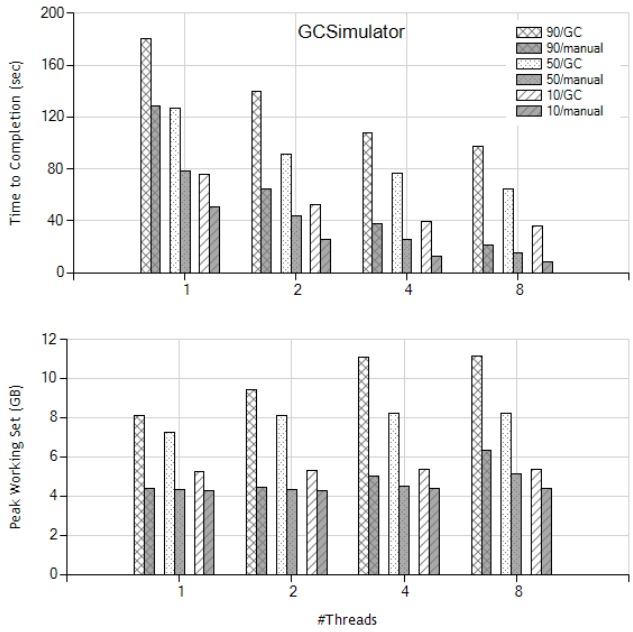


Figure 6: Time to completion and peak working set of the GC simulator benchmark with garbage collection and manual memory management with varying number of threads and percentage of short lived objects.

Compiler transformations. The compiler transformations described in Section 2.5.2 increase the lifetime of variables containing references to parent objects and introduce additional instructions. We measured the effect of these transformations in isolation by comparing the performance of our benchmarks with a garbage collector, with and without transformations enabled. Our experiments show that the transformations have no measurable impact on performance.

GC simulator. We configured the benchmark to create long-lived binary trees with a total of 2^{23} (little over 8 million) objects divided equally between threads. Short-lived and long lived are stored inside the tree and trigger replacement when visited 3 and 10 times respectively. Each thread makes 10 passes over its tree. We vary the number of threads, size of the payload in each object (16, 64 and 256 bytes) and the fraction of short lived objects (10, 50 and 90%).

Figure 6 shows the time to completion and peak working set both with GC and manual memory allocation for 256-byte objects. With the garbage collector, the benchmark does not scale linearly, especially when the ratio of short lived objects (and hence the allocation rate) is high (speedup of 1.85X with 8 threads) in spite of the benchmark being data parallel. Also observe that the peak working set is higher when the fraction of short-lived objects is higher and increases with the number of threads. Figure 6 also shows the execution time and working set of the benchmark using our allocator. The benchmark scales almost linearly with the number of threads (3.45X with 4 threads and 5.96X with 8 threads). Also observe that the peak working set is much lower ($< 1/2$ compared to GC in some configurations) and remains almost constant irrespective of the allocation rate and the number of threads. This is an artifact of our allocator's policy to trigger promotion once the degree of fragmentation exceeds 0.5.

To understand this difference in performance, we measured a number of GC and allocator related metrics for this benchmark (Table 2). As expected, the number of *gen0* collections increases with the fraction of short-lived objects, independent of the number of threads. The maximum pause time increases with the fraction of short lived objects (up to 8.9 seconds), while the average pause time remains more or less unchanged. However observe that the percentage of time the application is paused for GC increases significantly with the number of threads, reaching up to 80%. This explains why this benchmark scales poorly with the GC even though it is data parallel.

Our allocator is able to recover the performance loss by eliminating long pauses. As shown in Table 2, our alloca-

Threads	% Short lived	gen0	gen1	gen2	% Pause for GC	Max pause (ms)	Avg. pause (ms)	#Promotion	#AV	#AV patched	#Dedangle
1	10	477	379	10	37.6	257	36	1	194449	165512	0
1	50	771	542	12	45.4	5644	49	5	2252247	1593650	5
1	90	1069	705	14	47.5	6884	52	10	6314512	4823422	7
2	10	557	308	8	47.1	212	29	1	223113	189809	0
2	50	859	473	9	56.1	339	41	4	2618080	1865863	4
2	90	1165	638	12	62.1	7468	51	9	5554439	3850610	8
4	10	560	306	4	63.3	390	31	1	249862	211691	0
4	50	864	472	6	67.5	695	41	4	3059286	2426542	3
4	90	1171	636	6	69.5	8588	51	5	5969741	4702079	4
8	10	562	306	3	75.4	885	30	1	49699	41862	0
8	50	866	470	3	74.6	859	41	3	1859671	1592611	0
8	90	1173	635	4	80.3	8921	48	3	2998572	2545600	0

Table 2: Execution profiles with the garbage collector and our allocator for the GC simulator using 256 byte objects with varying number of threads and percentage of short lived objects. % Pause GC is the percentage of time the application was paused for garbage collection. #AV is the number of access violations encountered and #AV Patched is the number of access violations patched lazily.

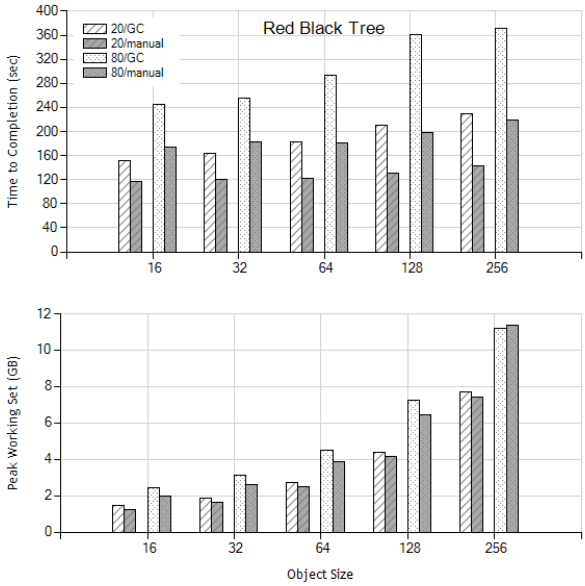


Figure 7: Time to completion and peak working set of the red-black tree for varying payload size and fraction of updates.

tor triggers promotion relatively infrequently (between 1 to 10 times). Furthermore, when an AV is triggered due to promotion, the allocator is able to patch parent references in a majority of cases. In some configurations, promotion causes a spurt of access violations, forcing the allocator to trigger de-dangling.

We also ran a variant of this benchmark where all short-lived objects are ephemeral and hence do not survive *gen0* collection. This variant is designed to compare raw allocation throughput of the GC vs. our allocator. We find that our throughput in this scenario is marginally lower due to system calls overheads of reclaiming memory.

Red black tree. We created a tree with 10 million nodes and executed 10 million operations while varying payload size and the fraction of updates. In this benchmark, switching to manual memory management improves execution

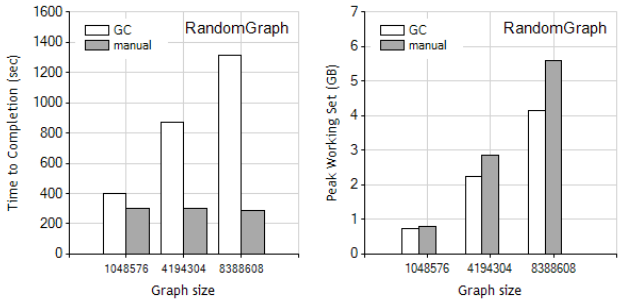


Figure 8: Time to completion and peak working set for RandomGraph with different graph sizes.

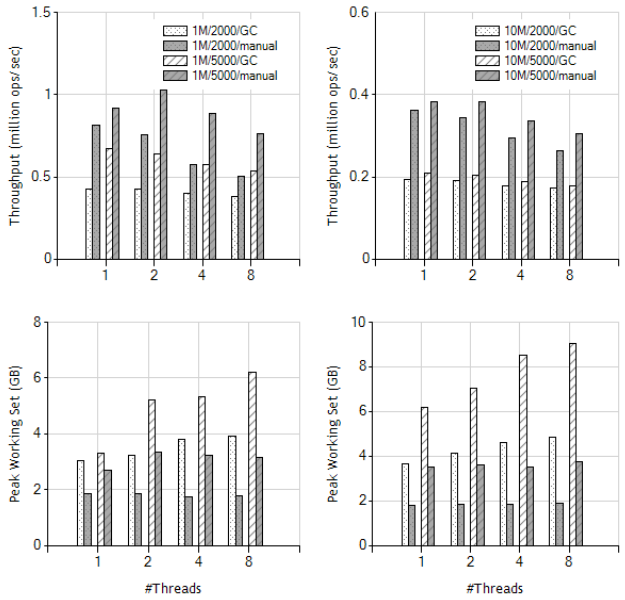
time by 38% on average (Figure 7). The speedup is higher for larger objects and when the fraction of updates is higher (which translates to an increased allocation rate). We observe that when the ratio of updates is small, the allocator never triggers promotion because the degree of fragmentation remains below the threshold through the execution. In contrast, the garbage collector must initiate collections with the hope of finding freed objects. Also observe that the reduction in memory utilization is not as significant. This is because almost all allocated objects in this benchmark have long lifetimes. With the GC, these objects are quickly promoted to *gen2*, which also expands to include all live objects in the tree.

Random graph. We configured this benchmark to create graphs with different number of vertices and connectivity and perform 10 million random replacements on the graph. This workload has a high rate of allocation and garbage generation. Therefore, the garbage collector triggers collections very frequently (Table 3). Our allocator also triggers promotions relatively frequently, especially for smaller graphs because the degree of fragmentation increases faster. Promotions result in a spurt of access violations due to high in-degree of vertices, and eventually trigger de-dangling

Figure 8 shows the execution time and peak working set for this benchmark for different graph sizes. We observe an improvement in throughput when using manual

Graph size	gen0	gen1	gen2	% Pause for GC	Max pause (ms)	Avg. pause (ms)	#Promotion	#AV	#AV patched	#Dedangle
1048576	579	231	9	73.1	773	437	20	32750481	32750455	20
4194304	709	321	13	80.0	3160	680	5	26587731	26587729	5
8388608	882	451	14	81.8	7323	812	2	14715787	14715786	2

Table 3: Execution profiles with the garbage collector and our allocator for the RandomGraph benchmark.



(a) 1M entries. (b) 10M entries.

Figure 9: Throughput and peak working set for ASP.NET caches with different sizes and different expiration times.

memory management. Perhaps surprisingly, we observe that the throughput remains constant independent of the graph size, which is ideal since the amount of work done remains the same. With garbage collection, throughput drops as the graph size increases. This is due to higher number of *gen0* and *gen1* collections that trace the entire graph to find free objects. We also find that the peak working set is higher with our allocator, especially in large graphs. For large graphs, our allocator triggers promotion less frequently.

ASP.NET caching. ASP.NET caching supports several configuration parameters that can be used to trade off hit rates with memory consumption. For our evaluation, we chose two different cache sizes (1M and 10M key/value pairs) and expiration policy (sliding window with a 2 seconds and 5 seconds). We created a multi-threaded client that randomly generates keys from a exponential distribution (with $\lambda = 1.5$) to model skew, issues a get requests and attempts to add a new value (randomly sized between 16 bytes and Kb) if no value exists. Using a skewed distribution results in contention due to threads acquiring writer locks and inserting values in a small part of the key space.

As shown in Figure 9, the overall throughput of the cache decreases as the number of client threads increases. This is

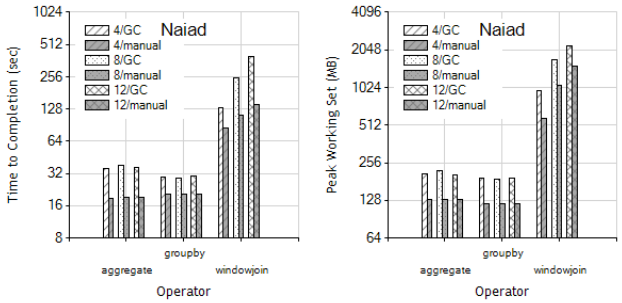


Figure 10: Time to completion and peak working set for three Naiad operators and varying window sizes.

expected due to contention. Also observe that throughput are higher with higher expiration times. This is because entries stay in the cache for longer resulting in higher hit rates. A cache with 1M entries achieves a hit rate of approximately 89% and 92% for expiration periods of 2s and 5s respectively.

Switching to manual memory management improves the throughput across configurations and significantly reduces memory consumption (by a factor of 2). Also observe that peak working set remains mostly the same independent of the number of threads. We also observe that in this application, the degree of fragmentation is low and never exceeds the threshold of 0.5. Therefore, the allocator never triggers a promotion phase. This is because objects that are allocated together (in time and space) tend to have the same lifetime. Therefore, entire pages are unmapped from *gen0* without promoting any objects.

Naiad. We executed queries over a streaming workload containing documents and authors. Each document has an identifier, a title, a length and an author, and each author has an identifier, name and age. The stream of document is divided into *epochs* with each epoch containing between 100000 and 500000 documents. Epochs in the author stream contain anywhere between 1 and 5 authors picked from a collection of 10 authors. For the evaluation, we picked three queries, an aggregation of document lengths by author, a grouping of documents by author, and a windowed join of documents and authors.

Figure 10 shows the execution time and peak working sets for three Naiad queries for different window sizes (in log scale). Aggregation and grouping are simpler operators that maintain relatively compact summaries of input streams. Therefore, these operations have a much lower memory footprint (and lower execution times). The windowed join on the

other hand maintains a much larger amount of state, which increases with window size. In all cases, the implementations that use our allocator outperform the ones with GC (by up to 3X) both in throughput and memory consumption.

RaptorDB. We use a stress test for RaptorDB that inserts 20 millions key/value pairs in the database, builds an index, and then performs 20 millions get requests over the entire key space. With the GC, RaptorDB achieves a throughput of 66000 ops/sec and has a peak working set of 2.2GB. With manual memory management, the throughput improves marginally to 69000 ops/sec (an improvement of 4.5%) and the peak working set reduces to 1.4GB. Further profiling suggests that this benchmark is predominantly IO bound and spends only 14% of its time in GC.

5. Related work

Several languages propose static type systems for manual memory management. Some of these languages are based on regions [10, 20, 40, 43]. For example, Cyclone [22, 38] integrates a garbage collector with safe manual memory management based on regions. Several languages propose using unique pointers to objects [5, 23, 29, 31, 38]using concepts from linear types [8, 41]; and borrowing of references for temporary use [11, 12, 42]. Languages with ownership types [10, 12, 13] and alias types [37] can express complex restrictions on object graphs. Capability types [43] can be used to verify the safety of region-based memory management. We propose a simpler programming model that neither requires any of these concepts nor imposes restrictions on pointer aliasing, concurrent sharing of pointers, or allocation and deallocation sites.

Several systems use page-protection mechanism to add temporal safety to existing unsafe languages: [3, 4, 27] allocate a different virtual and physical page for every allocation; [16] improves efficiency by sharing a physical page for different virtual pages. However, it suffers from increased `TLB` pressure and high-overhead for allocation. We propose techniques that eliminate these limitations and achieve high performance.

Some systems propose weaker guarantees for safe manual memory management. Cling [6] and [17] allow reuse of objects having same type and alignment. DieHard(er) [9, 35] and Archipelago [27] randomize allocations to make the application less vulnerable to memory attacks. Several systems detect accesses to freed objects [26, 32, 44], but do not provide full type safety.

Several systems have proposed optimizing garbage collection – for instance for big data systems [15, 18, 28, 34] and for real-time and embedded systems with very low latency constraints [7]. Rather than optimizing garbage collection, we propose replacing it with manual memory management. Another line of work uses page-protection to eliminate pauses during compaction [14, 39]. C4 compacts relocates fragmented pages to unused pages. However, C4 uses a read barrier whenever a reference is loaded from memory to detect and patch invalid references. A loaded value barrier ensures that at any point all the visible loaded references can be dereferenced. Pauseless [14] uses a special hardware to implement this, whereas C4 [39] emulates the read barriers in software. Our allocator allows invalid references to be loaded and copied; these references are patched only when they are dereferenced. We use compiler transformations to achieve this efficiently.

Compressor [25] uses page protection for pause-less compaction. Unlike [14, 39], Compressor does not require read barriers, but can only compact the entire heap at once. Unlike these systems, we do not require tracing the object graph through potentially very large heaps to discover objects that can be freed; instead we rely on programmers explicitly deallocating objects.

Scala off-heap[4] provides a mechanism to offload all allocations in a given scope onto the unmanaged heap but does not provide full temporal and thread safety. Related work in the .NET ecosystem is Broom [19] that relies on regions but does not offer type safety. Recent recent work has also proposed the use of hints to allocate objects in arenas [34] without sacrificing safety. Safety is ensured using write barriers to track cross arena references, and objects are migrated from one arena to another or to the GC heap when programmer hints are wrong.

6. Conclusion and Future work

We presented a design for simple and fast manual memory management in safe languages. We propose simple programming model that does not impose any static constraints. Experiments show that this design achieves up to 3X reduction in memory and run time. An interesting direction for future work is to integrate manual memory management in languages with garbage collection, giving developers the choice of exercising control where desired while benefiting from automated memory management at the same time.

References

[1] Fundamentals of garbage collection, . URL `https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx`.

[2] .NET native and compilation, . URL `https://msdn.microsoft.com/en-us/library/dn807190(v=vs.110).aspx`.

[3] Electric fence malloc debugger. `http://elinux.org/Electric_Fence`.

[4] How to use pageheap utility to detect memory errors. `https://support.microsoft.com/en-us/kb/264471`.

[5] Rust programming language. `https://www.rust-lang.org`.

[4] `https://github.com/densh/scala-offheap`

[6] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[7] D. F. Bacon, P. Cheng, and V. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, 2003.

[8] H. G. Baker. Use-once variables and linear objects–storage management, reflection, and multi-threading. *SIGPLAN Notices*, 30(1):45–52, January 1995.

[9] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[10] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI*, 2003.

[11] J. Boyland. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience*, 31(6):533–553, 2001.

[12] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, July 2003.

[13] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, October 1998.

[14] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56. ACM, 2005.

[15] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. Idle time garbage collection scheduling. In *PLDI*, 2016.

[16] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN*, June 2006.

[17] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, 38(7):69–80, 2003.

[18] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: a garbage collector for big data on big NUMA machines. In *ASPLOS*, 2015.

[19] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *HotOS*, 2015.

[20] D. Grossman, G. Morrisett, and T. Jim. Region-based memory management in Cyclone. In *PLDI*, 2002.

[21] M. Hertz and E. D. Berger. Quantifyng the performance of garbage collection vs. explicit memory management. In *OOPSLA*, 2005.

[22] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *ISMM*, 2004.

[23] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.

[24] R. Hundt. Loop recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011.

[25] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, 2006.

[26] B. Lee, C. Song, Y. Jang, and T. Wang. Preventing use-after-free with dangling pointer nullification. In *NDSS*, 2015.

[27] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS*, 2008.

[28] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, 2016.

[29] N. Minsky. Towards alias-free pointers. In *ECOOP*, pages 189–209, July 1996.

[30] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, 2013.

[31] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, 2012.

[32] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS compiler-enforced temporal safety for c. In *ISMM*, 2010.

[33] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, 2015.

[34] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high performance big-data-friendly garbage collector. In *OSDI*, 2016.

[35] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[36] R. Sedgewick. Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures*, page 17, 2008.

[37] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, 2000.

[38] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory-management in Cyclone. *Science of Computer Programming*, 62(2):122–14, October 2006.

[39] G. Tene, B. Iyengar, and M. Wolk. C4: The continuously conucrrent compacting collector. In *ISMM*, 2011.

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

[41] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference*, 1990.

[42] D. Walker and K. Watkins. On regions and linear types. In *ICFP*, 2001.

[43] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 24(4):701–771, 2000.

[44] Y. Younan. FreeSentry: protecting against user-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[45] B. G. Zorn. The measured cost of conservative garbage collection. *Software – Practice and Experience*, 23(7):733–756, 1993.