Leaky regions: linking reclamation hints to program structure

Tim Harris
Microsoft Research Cambridge
tharris@microsoft.com

Abstract

This paper[1] presents a new mechanism for automatic storage reclamation based on exploiting information about the relationship between object lifetimes and points in a program's execution. In our system method calls are annotated to indicate that most of the objects allocated during the call are expected to be unreachable by the time it returns. A write barrier detects if objects escape from one of these calls, causing them to be retained and subsequently managed by an ordinary generational collector. We describe a tool that helps select suitable annotation sites and we outline how this process can be fully automated. We show that if these annotations are placed judiciously then the additional costs of the write barrier can be outweighed by savings in collection time.

1. Introduction

In many cases there is a clear relationship between points during a program's execution and sets of objects that can safely be deallocated. For instance, a video-playback application may generate temporary objects while decoding one frame and most of these objects can be deallocated when moving to the next frame. Or a chess-playing game may allocate a number of objects while evaluating possible moves and discard most of these after choosing how to proceed.

If a tracing garbage collector is used with this kind of application then the cost of performing a collection will vary according to when the collector happens to run – it is usually better to invoke the collector when a large fraction of the storage space it will process contains unreachable objects. Schemes such as generational [21], older-first [18] and garbage-first [11] collection can be seen as different ways to select a good area of the heap to collect.

Currently, however, garbage collection is usually triggered without any reference to whether or not it is an appropriate time to perform a collection – for example, it may be faster to perform more frequent collections that are timed to coincide with low volumes of live data than to perform fewer collections which see a fuller heap.

In this paper we investigate a scheme for allowing the garbage collector to exploit information about the object lifetimes that occur in an application. Our approach is based on using annotations to indicate method calls in which most of the objects allocated during the call are expected to be unreachable by the time the call returns. These annotations are always safe – if they are placed injudiciously then the program may run slower than before but it will not crash.

Entering an annotated call creates a new *scoped region* in the heap within which subsequent object allocation occurs. As with generational collection, a write barrier is used to detect objects that may escape from a scoped region. Each time an annotated call returns we garbage collect the associated region, promoting any escaping objects into the next youngest region (in the case of nested annotated calls), or into the young generation of the heap (in the case of the outermost annotated call). Figure 1 illustrates this general scheme and Section 2 describes this design in more detail.

In practice we expect these annotations to be produced from profiling, although expert programmers may choose to manually annotate code. Others have developed suitable profiling systems which could be used [5, 14, 9] and so we only briefly mention our profiler in Section 3.

We have prototyped this design as part of a research compiler and run-time system for executing CIL (Common Intermediate Language) programs for .NET. As with the Jikes RVM [1], the vast majority of the run-time system is itself implemented in safe bytecode. For readers familiar with Java bytecode but not CIL, it is worth pointing out that the only significant difference to consider here is that the safe subset of CIL allows indirect stores to older locations on the current thread's stack.

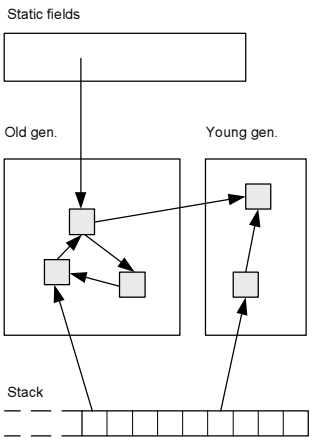
We have tested our implementation with a number of benchmarks compiled from C# to CIL. In the benchmarks we have studied, the effective placement of annotations led to a *collect early, collect often* profile of memory usage: many more scoped memory regions are created and reclaimed than the number of garbage collections that would be triggered with the same heap size. We discuss performance results in Section 4.

Our decision to focus on using a write barrier to check for escaping objects may appear perverse given recent work on escape analysis and region-based storage management. However, although dynamic checks add work on the write barrier, our approach gains a number of advantages over static analyses. Firstly, it allows us to safely allocate objects which are unlikely to be identified as non-escaping by a static analysis. Secondly, it allows us to optimistically scope-allocate objects even when similar objects occasionally escape (for instance, if objects escape when an exception is raised, but not ordinarily). Finally, by using an appropriate form of write barrier, we can reclaim objects that are dead at the end of a scoped region but which temporarily escape during its execution (for instance, in one of our benchmarks, by being temporarily reachable from a static field).

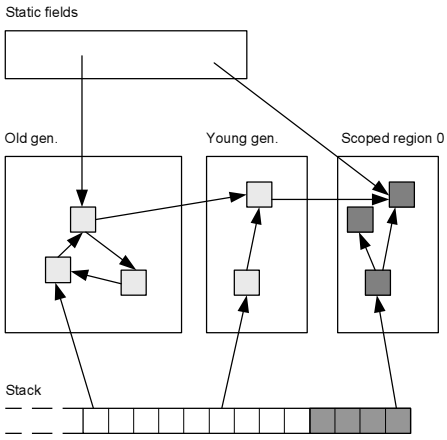
We discuss related work in more detail in Section 5 but we should emphasise that the underlying idea of combining optimistic stack allocation with garbage collection is not new. For instance, Baker argued that allocations should often be made on the stack and lazily promoted to the heap [2]. More recently, stack-like object management disciplines have been used in ML [19] and in the Real-Time Specification for Java [4]. Qian *et al.* [16] and Corry [9] independently assessed how similar styles of allocation could be used in object-oriented languages. The key contribution of our work over Qian *et al.*'s and Corry's is that we show how to actually implement the scheme efficiently at run-time – getting measured improvements on non-trivial benchmarks in terms of running time as well as in terms of space reclamation.

A weakness of our current system is that it is restricted to single threaded applications. We discuss the consequences of extending our design to support multi-threading in Section 6.

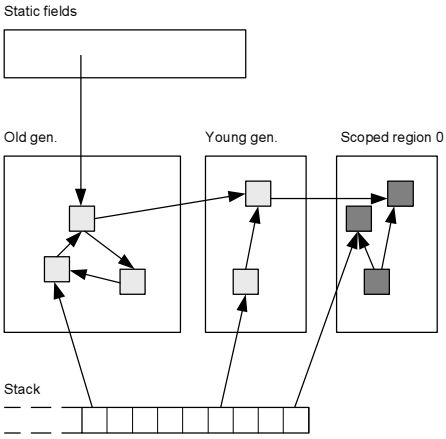
[1] This paper describes work undertaken July 2004 – December 2004.



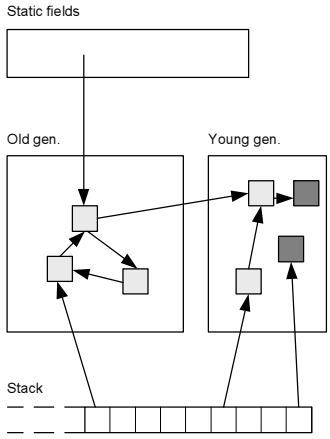
1. The heap state just before entering a function for which a new allocation scope is to be created.



2. Within the function, objects are placed in a new region of the heap. References to these objects can be manipulated freely, e.g. they can be referred to from the stack, from other heap regions and from static fields.



3. Upon returning from the function, some of the objects allocated in it may still be reachable, both from the heap, from statics and from return values and exceptions on the stack.



4. Before deallocating the scoped region, the reachable objects are promoted into another region and references to them fixed up: in this case they have been promoted into the young generation.

Figure 1. Overview of the creation, use and deallocation of a new allocation scope.

2. Design

After presenting out design goals in Section 2.1, we discuss the original design of the heap in our run-time system (Section 2.2), how we extend it to represent scoped regions (Section 2.3), how we use the write barrier to track inter-region references (Section 2.4), how we decide when to reclaim space from scoped regions (Section 2.5) and finally we look at a number of implementation issues (Sections 2.6–2.8).

2.1 Goals

A number of goals have guided our design and implementation work:

Safety. We do not want to provide mechanisms by which dangling references could be created. We also do not want to introduce new kinds of exceptional failure, for instance if an unexpectedly deep nest of annotated calls is executed, or if a large volume of objects escapes from a scoped region.

This goal is important because it enables annotations to be added based on intuition or profiling of common-case code without

```
void a() {
  // Allocation
}

void b() {
  while (..) {
    a(); // Annotated call
  }
}

b(); // Annotated call
```

```
void c() {
  // Allocation
}

void d() {
  c(); // Annotated call
  while (..) {
    // Allocation
  }
}

d(); // Annotated call
```

Figure 2. The left hand code fragment contains an annotated call to `b` within which a loop makes a series of many annotated calls to `a`. Being able to collect the storage space allocated by `a` without triggering an ordinary GC requires the *inner* calls to `a` to be supported. Conversely, the right hand code fragment contains an annotated call to `d` within which a single annotated call to `c` is made, followed by a loop which performs a number of allocations. Being able to reclaim this storage without an ordinary GC when `d` returns requires us to retain information about the *outer* of two nested scopes.

needing to consider rare execution paths or the effects of not-yet-loaded classes.

Nesting and composability. We want to support nesting between annotated calls – arising either through recursion, or by calling in to library code that uses annotated calls internally.

Figure 2 illustrates why the simple design of providing a single scoped region as some kind of "younger than young" generation would be ineffective by showing examples of nesting where the inner of two scopes should be retained and, conversely, where the outer of two is more appropriate to honour.

Low fragmentation cost. In order to support nesting effectively, we must avoid fragmentation costs when entering new annotated calls – for instance, it would be inappropriate to place allocations in different regions on different virtual memory pages.

Low execution overhead. The overheads introduced on executing an annotated call, on executing write barrier code to detect escaping objects, and on leaving an annotated call in the case where nothing escapes must not offset the performance gained by reducing the time spent in garbage collection. A further consideration is the need to avoid spending extra time clearing memory now that it is being reclaimed when existing annotated calls return and therefore at smaller granularities than when completing an ordinary GC.

Allow recapture. It should be possible to deallocate objects that temporarily escape but are recaptured by the time that the annotated call returns (for instance, because they are temporarily reachable from a static field). We see this kind of object usage in the Lisp interpreter benchmark in Section 4.

Interact with full language features. We should support all of the features of the CIL bytecode language and core libraries – including reflection, multi-threading, weak references and finalization. However, our current design and implementation does not address these problems; we return to discuss them in Section 6.

2.2 Original heap design

In the experiments performed here, the heap used in our run-time system has a conventional two-generation stop-the-world design. Young generation collections are performed using a semi-space collector that is usually triggered after a fixed volume of allocation. A full collection, using a sliding collector, is invoked every 8 young generation collections.

```
int a(int n) {
  // Temporary allocations

  if (n < 5) {
    t = a(n + 1); // Annotated call
  }

  // Temporary allocations

  return t;
}

r = a(0); // Annotated call
```

Figure 3. Within the annotated call to `a(0)`, a nested series of scoped regions are created, with objects being allocated within each one during the recursive calls, some of which escape to the caller.

Heap storage is divided into aligned 4KB pages and, for each page, an ownership byte is held. For pages containing objects, this indicates to which generation the objects belong. Other ownership values distinguish (*i*) pages holding static data, (*ii*) pages forming part of threads' stacks, (*iii*) pages holding non-heap data (such as code), (*iv*) empty pages which have been cleaned and (*v*) empty pages which have not been cleaned.

Ordinary allocations are satisfied from thread local allocation buffers, with the fast path code for non-overflowing allocations being inlined by the compiler. Large objects (64KB+) are allocated directly in the old generation.

A write barrier tracks old-to-young references, using the page ownership table to confirm that an update is creating such a reference before logging the updated location in an sequential store buffer (SSB). Each thread has a chunk of SSB space; if the supply of SSB chunks is exhausted then a young generation collection is triggered. We discuss, along with our results, the volume of SSB entries logged. Thread stacks and statics are treated as GC roots and so, using a stop-the-world collector, updates to them do not require tracking by the write barrier.

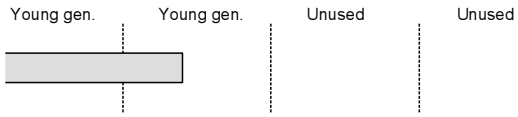
2.3 Representing scoped regions

There are two main decisions to take in how to represent scoped regions in the heap: how to allocate objects in different regions and how to identify inter-region references which may cause objects to escape the region in which they are allocated.

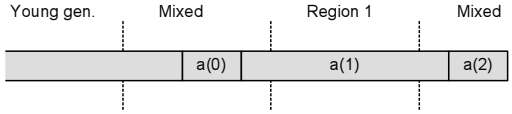
The approach we take for allocation is to retain the same local allocation buffer that was active on executing an annotated call and to introduce a new set of page ownership values for scoped regions. There are a number of trade-offs involved here:

- Our design makes entering a scoped region fast – we do not need to refill the local allocation buffer.
- Annotated calls do not cause fragmentation – in particular a deep series of calls does not lead to many unused fragments in the way that it would if we used fresh local allocation buffers for each.
- We lose precision in the ownership information retained about pages – we must introduce a `mixed` ownership value for pages containing objects from more than one scoped region, or from the young generation as well as the outermost scoped region.
- We are unable to allocate objects within anything but the innermost region – for instance, a method cannot directly allocate an object to be returned in the region corresponding to its caller.

As an example, consider the example method call `a(0)` depicted in Figure 3. This makes recursive calls `a(1)`... `a(5)`, allocating



1. Initially heap pages are being used to hold objects in the young generation.



2. After calls to `a(0)` and recursively to `a(1)` and `a(2)` three scoped regions are active. The ownership table records pages holding objects from different scoped regions as `mixed` (in practice regions would span many pages).

Figure 4. Heap page usage on entering scoped regions.

some objects in each call. Figure 4 shows how these objects may be laid out across a number of pages in the heap and the ownership values that will be associated with these pages.

Note that although the figure shows only four pages, we actually expect scoped regions to be from around 1KB to 1MB in size. This is because regions smaller than this are unlikely to be effective unless the entry/exit cost can be amortized over a larger volume of allocation. Conversely, as regions grow beyond 1MB, it becomes more likely that they will span GC cylces. Our profiling results in Section 4 show that annotations for regions of 1KB-1MB can be found in practice.

2.4 Write barrier

Without scoped regions, the write barrier need only track references from the old generation (ownership value 1) to the young generation (ownership value 0). Negative ownership values are used for pages holding code, statics, thread stacks and non-heap data (such as the ownership value of `null` references). In pseudo-code the write barrier executed for a store `*a = r` is:

```
void write_barrier(void **a, void *r) {
  char owner_a = GET_OWNERSHIP_VALUE(PAGE_OF(a));
  char owner_r = GET_OWNERSHIP_VALUE(PAGE_OF(r));
  if ((owner_a > owner_r) && // Store is old-to-young
      (owner_r > 0))         // Stored value is a ref
  {
    SSB_LOG(a);
  }
}
```

In practice the ownership lookups and comparison are inlined and the `SSB_LOG` operation is kept out of line. The bytecode-to-native compiler can perform common-subexpression elimination between `GET_OWNERSHIP_VALUE` operations (e.g. to avoid re-fetching the table involved).

To support scoped regions we need to track further kinds of store which could let objects escape: *(i)* references from objects in the ordinary generations to objects in scoped regions, *(ii)* references from statics to objects in a scoped region, *(iii)* references from stack frames outside an annotated call to objects instantiated during the call. There is a tension between these requirements – making the first decision precisely may add complexity to the write barrier, whereas the second and third add more barriers to the program and so makes it important to keep them fast.

We use two techniques to reduce the cost of write barriers. Firstly, as Stefanović has done [18], we organise the heap so that when executing a store `*a = r`, a comparison of the virtual addresses `a` and `r` can identify many cases where no further test is necessary[2]. Figure 5 shows the layout we used.

[2] In the performance results here we added a similar filter to the ordinary write barrier (providing a marginal improvement over the original system).

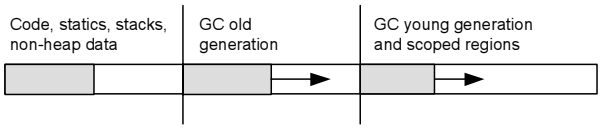


Figure 5. Virtual address space layout. References from right to left (higher addresses to lower addresses) never need to be tracked by the write barrier. References from left to right need to be tracked if they could let objects escape from scoped regions.

Secondly, we restructure the representation of ownership values. Instead of representing them by single integer values, we use a packed representation split into an *ownership kind* and well as an *ownership age*. The packed form means that 'kind' field is more significant than the 'age' field. Implementation limits, keeping the resulting packed value in a byte along with other flags, provide 4 bits for the age and 2 bits for the kind.

Figure 6 shows how these fields are used. Notice that the particular numerical values are carefully chosen so that:

- The ordering of ownership kinds means that static data and stack locations appear younger than the ordinary GC generations but older than scoped regions.
- The ordering of packed values will detect old-to-young references in the heap, references from statics and stacks to objects leaking from scoped regions, as well as references from older scoped regions to younger ones.
- The LSB of the ownership kind indicates if the page contains objects.

The result of this design is that we add a further test to the write barrier and change a comparison against zero with a mask:

```
void new_write_barrier(void **a, void *r) {
  char owner_a = GET_PACKED_OWNERSHIP_VALUE(PAGE_OF(a));
  char owner_r = GET_PACKED_OWNERSHIP_VALUE(PAGE_OF(r));
  if (((owner_a > owner_r) &&  // Store may be old-to-young
       (owner_r & 16))      // Stored value is a ref
      || (owner_r == 0x1f)) {  // Referent of unknown age
  {
    SSB_LOG(a);
  }
}
```

The write barrier does not have any information about which stack frames correspond to annotated calls – we assume that indirect stores into the stack are rare and so we choose to log all of those which refer to objects in scoped regions and filter them on deallocation. We have not observed this as a problem in practice.

Page kind	*Ownership kind*	*Ownership value*	*Packed*
Oldest GC generation	0x3	0xf	0x3f
Youngest GC generation	0x3	0x0	0x30
Code, statics, stacks	0x2	0x0	0x20
Mixed ownership	0x1	0xf	0x1f
Outermost scoped region	0x1	0xe	0x1e
Innermost scoped region	0x1	0x0	0x10
Unused or unallocated	0x0	0x0	0x00

Figure 6. Revised structure of ownership values. The *packed* column shows how the combined *kind* and *value* bit-fields compare numerically.

2.5 Region management

We now turn to a number of policy decisions about how to organise the creation and reclamation of scoped regions.

The first question is how to manage the limited number of scoped region ages that are available with the current form of ownership table – the problem is what happens if we try to execute an annotated call when we are already performing allocations within a region using the most deeply nested ownership value.

One option would be to 'flatten off' the nesting hierarchy at that point, so that once we reach the maximum depth, any subsequent annotations are disregarded. However, following on from the observations about nesting and composability in Section 2, we chose not to follow this path because it interacts poorly with hierarchies of nested calls in which the bulk of the allocation work occurs deep within the hierarchy.

Instead, if we exceed the maximum nesting depth, we *abandon* the current set of nested scopes and start again with the outermost ownership value. Our heap layout makes it straightforward to do this: we update the page ownership table for the existing scopes, marking all but the most recent page as part of the young generation, and marking the most recent page as unknown. Figure 7 illustrates this.

The second major point where a policy decision is needed is how to proceed when returning from an annotated call. In general there are two options: attempt to *reclaim* the region, or *defer* the region by merging it into its enclosing one or into the young generation. If we defer a region then dead objects in it will be considered for reclamation the next time we exit an enclosing region.

Primarily the reclaim/defer decision is a trade-off between the opportunity to reclaim storage space versus the computational effort required to do so safely. A secondary consideration, given appropriate annotations, is that deferring a region is appropriate if objects have escaped from it to an enclosing region – waiting until the enclosing annotated call returns will give these escapees a chance to be recaptured.

The default policy we use is based on statistics that are available quickly on returning from an annotated call. We defer only if:

- We are dealing with an enclosed scoped region rather than the outermost one – we always reclaim the outermost scoped region because a number of enclosed regions may have already been deferred into it.
- Some SSB entries have been logged during the region's execution (conversely, if no entries have been logged, then no objects can have escaped from the region and so it is trivial to reclaim it).
- We are still using the same local allocation buffer as when we started the call – this means that there is little potential benefit in reclamation at this time.

2.6 Deferring a region

As when abandoning regions, the heap structure means that deferring the collection of one region into its parent is largely a case of updating the page ownership table for the pages being transferred.

However, there is one subtle problem that occurs: we need to remove any SSB entries relating to stores in stack frames of methods that have now returned. This is necessary in case that memory is re-used for stack frames which store non-pointers at those addresses.

2.7 Reclaiming a region

At a high level, reclaiming a region on return from an annotated call involves preserving the objects transitively reachable from SSB entries that were logged during the call, along with the return value (if it is an object reference), or any exception object being raised (if, as usual, it was allocated during the call).

Following our assumption that few objects escape from annotated calls, we use a compacting scheme to avoid fragmenting the heap. For implementation simplicity, and to avoid a further pass over the space being reclaimed, we actually use a two-copy design, evacuating any escaped objects to fresh pages[3] before copying them back to the start of the memory that the region occupied; an ordinary mark-compact collector could of course be employed to avoid one of these copies.

Figure 8 illustrates the heap during these steps.

2.8 Interaction with garbage collection

Ordinarily, our run-time system collects the young generation after a given volume of allocation. To incorporate scoped regions, we instead collect whenever the total volume of space occupied by the young generation and the currently active scoped regions exceeds a given volume. This gives the same collection behavior when scoped regions are not in use, while allowing collection to be deferred while scope reclamation is allowing the same memory to be re-used. When a garbage collection occurs, we abandon the currently active scoped regions as described in Section 2.5.

3. Profiling

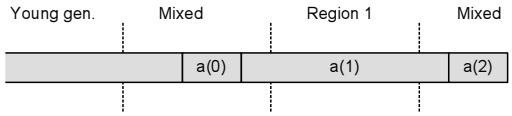
In practice we expect scoped regions to be identified from profiling although, of course, expert programmers may choose to manually annotate their code.

We built a simple profiling tool to identify plausible annotation sites. This tool instruments each function call site to record (*i*) the number of times that the call was executed, (*ii*) the total volume allocated during those calls, (*iii*) the maximum volume allocated within any one call. This does not show whether or not the allocated objects escape from the call, but it serves to eliminate calls which are either executed very frequently (where the cost of scope entry or exit would become important), within which small volumes are allocated (and so the possible benefits of reclamation are low), or where very high volumes of data are allocated in single calls (and so a garbage collection is likely to be triggered, causing the scoped regions to be abandoned).

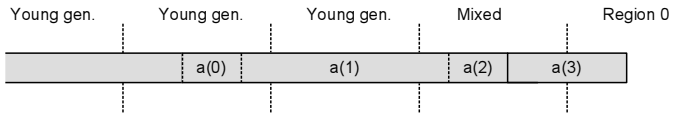
The volume of allocation recorded by the tool provides an upper bound on the volume that could be reclaimed by annotating the associated call site: the actual amount reclaimed will be reduced because (*i*) some of the allocated objects may escape, (*ii*) some of the allocation may be of large objects, placed directly in the old generation, (*iii*) the calls might nest so deeply that some of the scoped regions are abandoned, and (*iv*) an ordinary garbage collection may also cause the active scoped regions to be abandoned.

In our prototype we examined the call sites identified by the tool, ranked according to the volume of data allocated within them, and tried the highest ranked entries within which at least a few kilobytes and at most a few megabytes was allocated per call. This policy has the effect of excluding calls which are executed very frequently or very infrequently. As we point out when discussing related work, there is clearly scope for a fully automated method for producing annotations and there are many promising techniques for doing so.

[3] We back off to triggering an ordinary GC if such pages are not available.

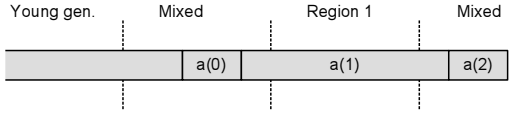


1. Initially, scoped regions 0..2 are associated with the calls `a(0)`, `a(1)` and `a(2)`.

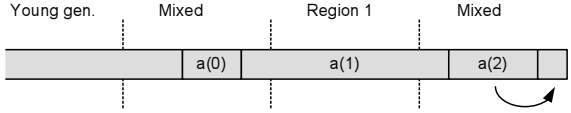


2. The scoped regions are abandoned by marking their pages as part of the young generation. Scoped region 0 can now be re-used for the call to `a(3)`.

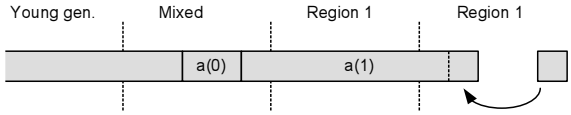
Figure 7. Abandoning scoped regions to re-use ownership values in nested calls.



1. Initially, scoped regions 0..2 are associated with the calls `a(0)`..`a(2)`.



2. The call to `a(2)` continues and eventually returns. Any objects escaping from region 2 are copied out.



3. The escaping objects are now copied back to be contiguous with region 1 for `a(1)`'s execution. We can now represent region 1's ownership more precisely.

Figure 8. Reclaiming storage allocated in scoped regions.

4. Performance

In this section we present performance results taken from allocation-intensive C# benchmarks modeled on the *ijpeg*, *go*, and *li* programs used in the SPEC CINT95 suite, and the *crafty* chess player.

Figure 9 shows the overall timing results of the four benchmarks when run with young generation sizes of 1MB, 2MB, 4MB, 8MB, and 16MB. We study a range of young generation sizes because the performance of the baseline garbage collector is highly dependent on the amount of memory in the young generation – the time spent in the garbage collector we use is, of course, highly dependent on the size of the young generation.

We break down execution time in terms of (*i*) time spent reclaiming objects (GC and scope management), (*ii*) time spent in storage management as a whole (GC, scoped region management, and clearing memory), (*iii*) total run time (including all these direct costs of storage management, plus the associated write barrier work in the mutator thread). The results presented are the median of 7 benchmark runs. We will consider the four benchmarks in turn, in increasing order of the volume of storage space that they allocate.

4.1 ijpeg

Profiling ijpeg showed that two calls are made from its `parse_args` method to `go_execute_compression_and_decompression` accounted for 29% and 31% of memory allocation, each being executed 64 times and the maximum volume allocated in any invocation being just over 200KB. A total of 29MB is allocated within these calls, of which 70KB escapes. The escaping objects are arrays which become reachable from fields of the `ijpeg` object.

Figure 10 shows the heap size while executing part of the benchmark. The sawtooth seen on the dark line shows how the heap is filled completely between young generations and how, when collections occur, they are timed at points where the heap contains live data. In contrast, collection using scoped regions occurs more frequently, but always when the heap is almost empty (the bottom points of the light sawtooth remain horizontal).

4.2 crafty

Profiling crafty showed that 217MB of the total 224MB allocation occurs within calls to `ABSearch` while selecting the next move to make. There are 84767 such calls and very few objects escape. The large number of calls means that a significant amount of reclamation time is spent in entering and leaving scoped regions, despite their effectiveness in replacing garbage collections.

On average around 2KB is allocated within each annotated call, meaning that many will complete without moving off a single page in the heap. However, there are a few calls which allocate large volumes, up to a maximum of 17MB, which cause young-generation GC to be triggered. This means that only 78% of storage space is reclaimed from scoped regions, even though 97% of objects are allocated during annotated calls.

The objects that escape are particularly interesting because the same allocation site generates escaping and non-escaping objects – the escapees

correspond to repeated board positions that are encountered while the game progresses[4]. Around 512KB escapes, comprising 13% of the objects allocated at the site in question.

4.3 go

Profiling showed that 709MB of the 732MB allocated during the go benchmark occurs within 296 calls to `getmove` from the main loop. None of these objects escapes because scalar fields are used to indicate the move that is selected. The high volume of allocation made within each invocation means that scoped regions only become worthwhile when the young generation is sufficiently large to contain an entire scoped region – the technique is effective only with 8MB and 16MB heaps.

4.4 li

The profiles observed for the Lisp interpreter are the most complex of the benchmarks we have tested. During the benchmark, the interpreter loads and executes a series of Lisp programs by calling the `xlload` method. Profiling revealed that 99% of storage was allocated within these calls. However, there are only 20 calls made, and thus numerous young generation collections would have occurred before a call returns. Another possible annotation would be a call to `evform` which also corresponds to 99% of allocation but which allocates only 50 bytes on average per call. However, this is inappropriate because of the high number of calls (41M) and the fact that the allocated storage is likely to be returned from the method.

We ultimately selected calls to `evfun` and `doloop` for annotation – the former covers 99% of allocation with 2.4M calls and the latter covers 64% with 0.1M calls. At run-time, we reclaim a total of 1.2GB and 0.8GB respectively – GCs occuring during `evfun` limit the effectiveness of that annotation and, of course, some nested calls involve both methods. The performance results in Figure 9 are particularly encouraging for 1MB and 2MB young-generation sizes. With 8MB and 16MB young generations, the cost of scope entry and exit dominates the cost of the garbage collections.

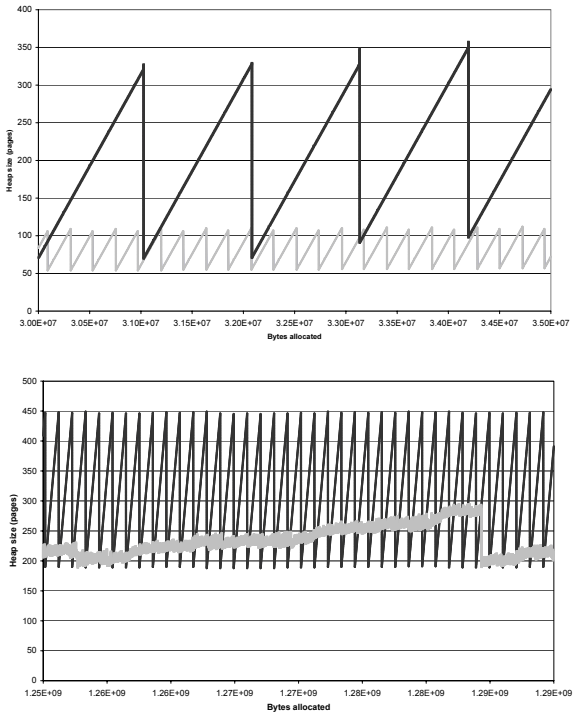


Figure 10. Heap size while executing part of *ijpeg* (left) and *li* (right), both with a 1MB young generation. The dark line shows the heap size solely using garbage collection and the light line shows the size using scoped regions as well.

5. Related work

Tofte and Talpin introduced the idea of using a stack of allocation regions as a mechanism for automatic storage management [19, 20]. In their original design, allocations can be made into any region in the stack and storage space is reclaimed by removing an entire region from the top of the stack when it is certain that none of the objects contained in the region will be accessed. For instance, temporary data used during a function's evaluation may be placed in a new region, but a data structure that is returned to the function's caller may be placed directly in the caller's region (or indeed in any older region).

They describe how region management annotations can be integrated into ML-like languages and a region inference system can be added to safely introduce these annotations automatically. The ability to allocate into regions at multiple levels would lead to fragmentation with the kind of sequential allocation used in our work and most run-time systems for object-oriented languages.

The Real-Time Specification for Java (RTSJ) contains a form of allocation region [4]. RTSJ describes this system in terms of run-time checks performed on assignment, raising an exception should an old-to-new reference be created in terms of region ages. This design is more restrictive than Tofte and Talpin's because it is expressed in terms of the shape of the object graph rather than the actual accesses made.

Deters and Cytron designed an algorithm for identifying RTSJ-style scoped regions in Java programs [10]. Their algorithm is based on execution traces and does not aim to be safe because the trace coverage may be incomplete. However, a variant of their algorithm could provide an excellent mechanism for automatically producing the annotations we require.

Cherem and Rugina designed a safe region analysis system for Java-like languages [7]. It allows non-lexically scoped regions and, unlike RTSJ, dangling references from regions where they can be shown to be safe. As with Deters and Cytron's design, combining static analysis with our own scheme to eliminate write barrier operations would be a promising direction for future work.

Qian and Hendren's work on dynamically identifying non-escaping objects using write barriers is similar to ours in that it allows allocations to be optimistically made in local regions, under the assumption that an object will not esacpe [16]. Their design works without any annotations or a statistics-gathering phase. However, it does so by associating a scoped region with every static call site; we have only ever seen good performance when annotations are placed judiciously. Their design also did not allow allocation sites classified as 'global' to revert to allocating in a local heap.

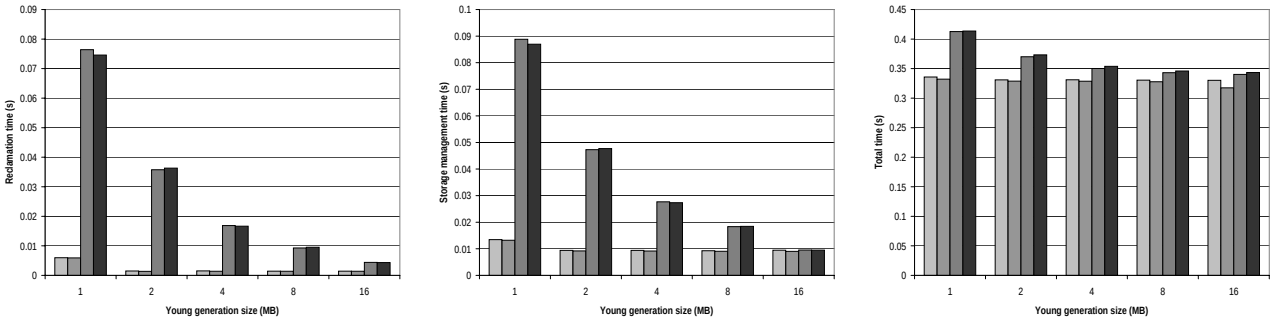
Corry's thesis, completed in parallel with our work, examines stack-based memory management in object-oriented languages [9]. Unlike our work, he associates scoped regions with loop iterations rather than with method calls. This fits with some plausible scenarios for stack-based memory management; ultimately we would expect a practical system to support both kinds of annotation (if only by re-factoring loop bodies into their own methods). Corry designed analysis techniques and a simulation framework for assessing different memory management policies (rather than an actual implementation). It would be promising to combine the results of that work with the practical run-time techniques that we have developed.

Buytaert *et al.* explored the link between program structure and GC costs [5]. They used offline profiling to identify favorable collection points (FCPs) at which the volume of live data is low. This is done using the Merlin algorithm [14] to collect traces of object lifetimes which can then be tied to method entry points. They saw substantial reductions in GC work in SPECjvm98 benchmarks because collection at FCPs is cheap and overall allows costly full-heap collections to be replaced by nursery collections. In some cases these benefits are due to the cyclic nature of the benchmark harness, but it is easy to imagine larger scenarios where the technique would be valuable.

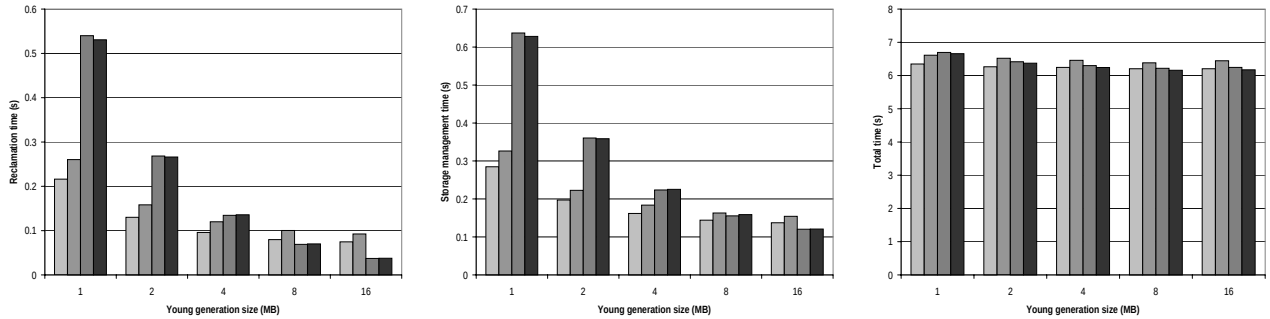
Buytaert *et al.*'s work is complementary to this paper because the two schemes operate at different timescales. Buytaert's results show that FCP-based collection allows GC cylces to coincide with program work, with a broadly unchanged number of GC cylces occuring. In contrast, our scoped regions are effective over much shorter timescales within a single collection cycle.

The use of escape analysis to enable on-stack allocation has often been proposed. This is an attractive proposition for simple cases. However, it runs into practical problems as well as those raised by the completeness of the

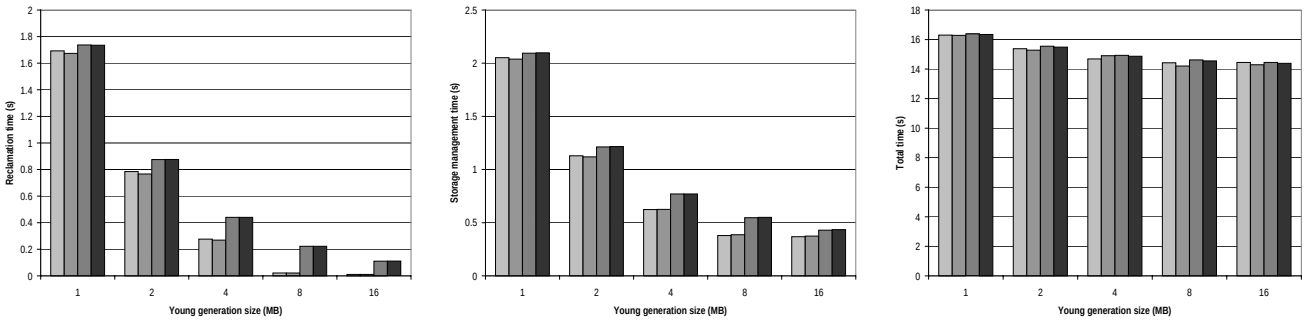
[4] The rules of chess say that a game is declared a tie if the same board position occurs more than three times.



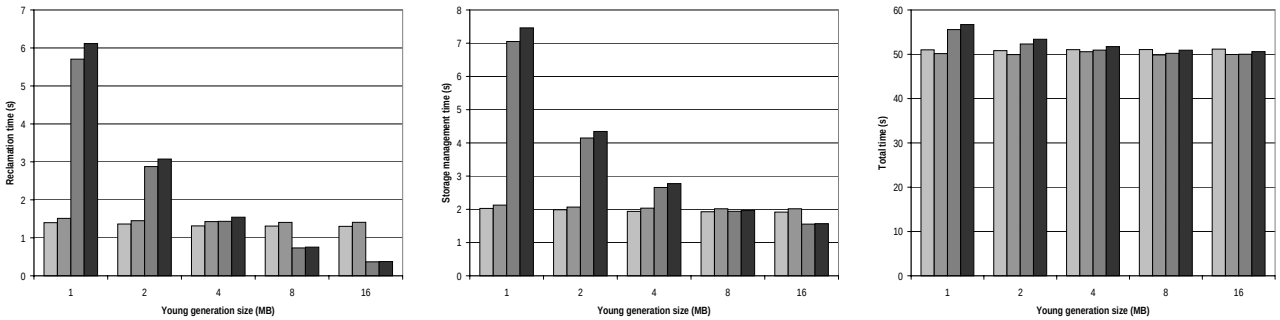
1. *ijpeg* (image compression and decompression) 45MB total allocation, originally 45 GC cylces (1MB young gen) .. 2 GC cylces (16MB young gen).



2. *crafty* (playing chess) 224MB total allocation, originally 224 GC cylces (1MB young gen) .. 14 GC cylces (16MB young gen).



3. *go* (playing go) 732MB total allocation, originally 732 GC cylces (1MB young gen) .. 45 GC cylces (16MB young gen).



4. *li* (Lisp interpreter) 2.1GB total allocation, originally 2062 GC cylces (1MB young gen) .. 128 GC cylces (16MB young gen).

Figure 9. Timing results from four benchmarks. Within each graph, the results shown in the lightest (leftmost) bar are using scoped regions as well as garbage collection with the defer/reclaim heuristic from Section 2.5, the second bar uses the simpler policy of *always* reclaiming storage, the third bar has the run-time support for scoped regions included but does not use any annotated calls, whereas the darkest (rightmost) bar is the original system using only garbage collection. The left hand four graphs record the time spent in storage reclamation (GC and scope management). The center four record the time spent in storage management (GC, scope management, and page zeroing). The right hand four graphs record total execution time.

static analyses used: space further up a traditional stack cannot be reclaimed and allocations in loops or during recursion may cause overflow.

Blanchet [3] and Choi *et al.* [8]'s recent papers discuss experiences in this area. Blanchet's results show that many objects can be stack allocated in small benchmarks such as Dhrystone. In larger programs they still see significant volumes of on-stack allocation, such as 43% (by volume) or 18% (by count) in a run of the `javac` compiler during which a total of about 7.5MB was allocated. Our results show much larger numbers of objects being allocated in scoped regions, due to a combination of (*i*) avoiding concern over stack overflow, (*ii*) being able to optimistically allocate objects that may escape. This gap suggests that a combination of stack-allocation and scoped regions would be effective: allocate on-stack where statically safe and low in volume, and use scoped regions otherwise.

Analyses have also been designed to enable objects to be allocated in thread-local storage. Domani *et al.* allowed objects to be optimistically managed as thread local (for instance, to elide synchronization) by associating a *global* flag with each object which is maintained by the write barrier [12]. Steensgaard's escape analysis is notable in that it can also handle some uses of static fields which do not allow objects to escape their creating thread [17]. Our design could readily be extended to multi-threaded programs for which this kind of analysis is effective; however, few object-oriented multi-threaded benchmarks currently exist to explore this direction.

6. Future work and conclusions

In this paper we have shown that a practical run-time system can be built to support dynamically-checked allocation regions and can significantly reduce the time spent in storage management even when used with optimised code, running applications that allocate large volumes of storage. Our results show that the technique is most effective when running with smaller young generation sizes – reclamation using scoped regions reduces or eliminates the need for full collection cycles, allowing memory to be reclaimed without scanning the threads' stacks. Coupled with the restriction to single-threaded use, the performance on small heaps suggests that scoped regions may be most appropriate for devices with limited physical memory and without parallelism in hardware – for instance managed code running on mobile phones. As our results show, larger heaps allow traditional garbage collection to operate efficiently (as one would expect given the asymptotic costs involved).

There are several directions in which we would like to develop this approach. Firstly, as we discussed in the previous section, we see this approach as complementary to work on stack-allocating objects; we can use scoped regions for objects that static analysis cannot prove as non-escaping but which profiling suggests are unlikely to escape in practice.

Secondly, we would like to extend our system to support multi-threaded applications beyond the simple uni-processor idea of abandoning the current scoped regions on a context switch. Although assuming single threaded execution is currently sufficient for many client-side applications, multi-processor machines are increasingly common as, of course, is concurrency within frameworks such as GUI systems.

There are two aspects to this problem. Firstly, our heap design assumes a simple linear ordering of the ages of different scoped regions, both in terms of the use of ownership values and in the address-based comparisons made in the write barrier. In a multi-threaded system, threads would enter and leave regions independently. Secondly, in the presence of concurrency, deallocating a region may not be safe if *any* of the objects contained in it have *ever* been reachable from other threads – for instance, if a reference to the object is assigned to a static field that is read by another thread, even if the field is subsequently overwritten by `null`).

Aside from the complementary static analyses for identifying thread-local objects, another promising direction is integrating dynamic techniques such as King's optimistic heaplets [15]. An initial combined design could introduce per-thread scoped regions within heaplets. Profiling would become particularly important to distinguish between objects escaping to other threads versus objects escaping to enclosing regions.

Finally, there is a synergy between our work on scoped regions and ongoing work on atomic transactions for shared-memory concurrency [13, 6] – allocation scopes which coincide with, or are contained within, transactional code can be treated as if they are single-threaded. That is safe because these systems rely on forms of optimistic concurrency control in which objects remain thread-local until their allocating transaction commits.

References

[1] ALPERN, B., ATTANASIO, C. R., BURTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHARD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeno Virtual Machine. *IBM Systems Journal 39*, 1 (2000), 194–211.

[2] BAKER, H. G. Cons should not cons its arguments, or, a lazy alloc is a smart alloc. *SIGPLAN Not. 27*, 3 (1992), 24–34.

[3] BLANCHET, B. Escape analysis for Java: Theory and practice. *ACM Trans. Program. Lang. Syst. 25*, 6 (2003), 713–775.

[4] BOLLELLA, G., BROSGOL, B., DIBBLE, P., FURR, S., GOSLING, J., HARDIN, D., TURNBULL, M., AND BELLIARDI, R. *The Real-Time Specification for Java*. Addison Wesley, June 2000.

[5] BUYTAERT, D., VENSTERMANS, K., EECKHOUT, L., AND BOSSCHERE, K. D. Garbage collection hints. In *HiPEAC 2005 International Conference on High Performance Embedded Architectures and Compilers, LNCS 3793* (Nov. 2005), pp. 233–248.

[6] CARLSTROM, B. D., CHUNG, J., CHAFI, H., MCDONALD, A., MINH, C. C., HAMMOND, L., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional execution of Java programs. In *Proceedings of the OOPSLA 2005 workshop on Synchronization and Concurrency in Object-Oriented Lanaguages (SCOOL)* (Oct. 2005). Also available in the University of Rochester digital archive `http://hdl.handle.net/1802/2096`.

[7] CHEREM, S., AND RUGINA, R. Region analysis and transformation for Java programs. In *ISMM 2004: Proceedings of the 4th International Symposium on Memory Management* (2004), pp. 85–96.

[8] CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst. 25*, 6 (2003), 876–910.

[9] CORRY, E. *Stack Allocation for Object-Oriented Languages*. PhD thesis, Department of Computer Science – Daimi, University of Aarhus, June 2004.

[10] DETERS, M., AND CYTRON, R. Automated discovery of scoped memory regions for real-time Java. In *ISMM 2002 Proceedings of the Third International Symposium on Memory Management* (Berlin, June 2002), pp. 25–35.

[11] DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. Garbage-first garbage collection. In *ISMM 2004: Proceedings of the 4th International Symposium on Memory Management* (New York, NY, USA, 2004), ACM Press, pp. 37–48.

[12] DOMANI, T., GOLDSHTEIN, G., KOLODNER, E. K., LEWIS, E., PETRANK, E., AND SHEINWALD, D. Thread-local heaps for Java. In *ISMM 2002: Proceedings of the 3rd International Symposium on Memory Management* (New York, NY, USA, 2002), ACM Press, pp. 76–87.

[13] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* (Oct. 2003), pp. 388–402.

[14] HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., MCKINLEY, K. S., AND STEFANOVIĆ, D. Error-free garbage collection traces: how to cheat and not get caught. *SIGMETRICS Perform. Eval. Rev. 30*, 1 (2002), 140–151.

[15] KING, A. C. *Removing Garbage Collector Synchronisation*. PhD thesis, University of Kent at Canterbury, September 2004.

[16] QIAN, F., AND HENDREN, L. An adaptive, region-based allocator for Java. In *ISMM '02: Proceedings of the 3rd International Symposium on Memory Management* (June 2002), pp. 127–138. A longer version of this paper appears as Sable Technical Report 2002-1.

[17] STEENSGAARD, B. Thread-specific heaps for multi-threaded programs. In *ISMM 2000: Proceedings of the 2nd international symposium on Memory management* (Oct. 2000), pp. 18–24.

[18] STEFANOVIĆ, D., HERTZ, M., BLACKBURN, S. M., MCKINLEY, K. S., AND MOSS, J. E. B. Older-first garbage collection in practice: evaluation in a Java Virtual Machine. In *MSP 2002: Proceedings of the workshop on Memory System Performance* (2002), pp. 25–36.

[19] TOFTE, M., AND TALPIN, J.-P. A theory of stack allocation in polymorphically typed languages. Tech. Rep. Computer Science 93/15, University of Copenhagen, July 1994.

[20] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation 132*, 2 (Feb. 1997), 109–176.

[21] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19*, 5 (Apr. 1984), 157–167.