# Memory Management with Explicit Regions

David Gay* and Alex Aiken*
EECS Department
University of California, Berkeley
{dgay,aiken}@cs.berkeley.edu

**Abstract**

Much research has been devoted to studies of and algorithms for memory management based on garbage collection or explicit allocation and deallocation. An alternative approach, region-based memory management, has been known for decades, but has not been well-studied. In a region-based system each allocation specifies a region, and memory is reclaimed by destroying a region, freeing all the storage allocated therein. We show that on a suite of allocation-intensive C programs, regions are competitive with malloc/free and sometimes substantially faster. We also show that regions support safe memory management with low overhead. Experience with our benchmarks suggests that modifying many existing programs to use regions is not difficult.

## 1 Introduction

The two most popular memory management techniques are explicit allocation and deallocation, as in C's malloc/free, and various forms of garbage-collection [Wil92]. Both have well-known advantages and disadvantages, discussed further below. A third alternative is *region-based* memory allocation, which has been widely used as an implementation technique for many years under a variety of names, e.g., zones [Ros67], groups [IY90], or arenas [Han90]. Regions have also recently attracted research attention as a target for static inference of memory management [TT97] and for improving locality of dynamically allocated data [Sto97].

In a region-based memory allocation scheme each allocated *object* is placed in a program-specified *region*. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. A simple example is shown in Figure 1. Each iteration of the loop allocates a small array. The call to `deleteregion` frees all arrays.

*This material is based in part upon work supported by NSF Young Investigator Award No. CCR-9457812, DARPA contract F30602-95-C-0136 and a Microsoft graduate fellowship.

```
void f()
{
  Region r = newregion();

  for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x);
  }
  deleteregion(r);
}
```

Figure 1: An example of region-based allocation.

In the commonly used version of region-based programming, regions are explicit in the program and entirely under programmer control. To our knowledge, the performance of this popular implementation technique has never been studied. Our first contribution is a detailed comparison of the performance of regions with malloc/free libraries and conservative garbage collection on a set of benchmark programs. Our conclusion is that explicit regions are, for our benchmarks, faster than either malloc/free or conservative garbage collection, and sometimes significantly so (up to 16%). Memory consumption is good in our experiments: regions use from 9% less to 19% more memory than the best alternative and always rank either first or second (see Section 5).

While our study supports the use of regions on performance grounds, the common implementation of regions is unsafe, as a region $r$ can be deleted even if other regions have accessible pointers to objects in $r$. Our second contribution is the study of a safe region implementation in which a region $r$ can be deleted only if there are no *external* references to objects in $r$ (a reference external to $r$ is any pointer not stored within $r$). We enforce this rule by keeping a *reference count* for each region; `deleteregion` is a no-op when this reference count is nonzero. Note that by reference counting regions instead of individual objects two common problems with reference counting are ameliorated: minimal space is devoted to storing reference counts, and cyclic structures can be collected so long as they are allocated within a single region. The overhead of safety varies from negligible to 17% on our benchmarks, but the comparative performance remains almost the same: regions are still faster (up to 9%) than the alternatives in all but a few cases, and in those cases regions are only slightly slower (up to 5%).

A third contribution is an assessment of how difficult it is to program with explicit regions. Our metric is the number and type of changes required to modify an application to use regions. All of our benchmarks required only modest recoding to use regions, and the needed region organization was straightforward to derive (see Section 5.1).

We also found that explicit regions have some particular strengths. First, regions bring structure to memory management, making programs clearer and, in our subjective assessment, easier to write compared to using malloc/free. For example, it is not necessary to walk through a complex data structure to deallocate it. Second, we found some evidence that regions can be used to provide significantly improved data locality, as posited by Stoutamire [Sto97]. The execution time of one benchmark was improved 24% simply by reorganizing allocation so that the most frequently accessed objects are allocated in a single region. Neither malloc/free nor garbage-collected systems provide any mechanism for expressing locality. Third, regions are useful for building software with predictable performance, as the cost of every operation is easily bounded, at least for unsafe regions and in [TT97] for compiler-inferred safe regions (see Section 2). Our safe implementation uses a moderately sophisticated scheme for efficiency; we show that the overhead of this scheme is amortized constant time per instruction executed, assuming that the size of stack frames is bounded by a constant (see Section 4.3).

Another advantage of region-based memory management is that it can be used nearly unchanged in an explicitly-parallel programming language. The only operations that require synchronization amongst all processes are region creation and deletion. Each process keeps a *local reference count* for each region which counts the references created or deleted by that process. A region can be deleted if the sum of all its local reference counts is zero. Writes of references to regions must be done with an atomic exchange (rather than a simple write) to prevent incorrect behaviour in the presence of data races, however the local reference counts can be adjusted without synchronization or communication.

There are situations where regions are not a good model, particularly when the programmer does not know enough about the lifetime of objects at allocation-time to place them in appropriate regions. One example we encountered is a game where objects are allocated and deallocated as the result of the player's actions; there is no way to place objects with similar lifetimes in a common region. Our purpose here is to study the costs and benefits of regions as they are normally used. We leave generalizations of explicit regions as future work.

We conclude this section with a high-level comparison of our region scheme with malloc/free and garbage collection. Our region model is reminiscent of malloc/free but allocation is about twice as fast and deallocation is much faster. In the safe version of our scheme, there is additional overhead for maintaining region reference counts. Garbage collection is easier to use than regions and can be very efficient if the application only uses a fraction of available memory. When an application needs most of the available memory, however, performance degrades. Also garbage collection prevents local reasoning about performance by introducing unpredictable pauses. Real-time collectors [Bak78, WJ93] eliminate this last problem at the cost of higher overhead. From these considerations, we believe regions are best suited for high-performance applications that use a large fraction of machine memory and where the lifetimes of values can be statically predicted. Regions are also useful for writing software with more predictable performance than garbage-collection-based systems.

The rest of this paper begins with a more detailed survey of related work (Section 2). We then introduce our safe, region-based memory management system (Section 3) and its implementation (Section 4). Section 5 details the costs of safe regions and compares their performance on six C applications with three malloc/free implementations and the Boehm-Weiser conservative garbage collector [BW88].

## 2 Related Work

The literature on memory management is vast. Surveys can be found in [Wil92] for garbage collection and [WJNB95] for explicit allocation and deallocation.

Regions have been used for decades. Ross [Ros67] presents a storage package that allows objects to be allocated in specific *zones*. Each zone can have a different allocation policy, but deallocation is done on an object-by-object basis. Vo's [Vo96] Vmalloc package is similar: allocations are done in *regions* with specific allocation policies. Some regions allow object-by-object deallocation, some regions can only be freed all at once. Hanson presents *arenas* in [Han90], but does not measure their performance. Barrett and Zorn [BZ93] use profiling to determine allocations that are short-lived, then place these allocations in fixed-size regions. A new region is created when the previous one fills up, and regions are deleted when all objects they contain are freed. This provides some of the performance advantages of regions without programmer intervention, but does not work for all programs. None of these proposals attempt to provide safe memory management.

Stoutamire [Sto97] adds *zones*, which are garbage-collected regions, to Sather [SO96] to allow explicit programming for locality. His benchmarks compare zones with Sather's standard garbage collector. Reclamation is still on an object-by-object basis.

The only published empirical studies on regions are for the region inference system of Tofte and Talpin [TT97], which automatically infers for ML programs how many regions should be allocated, where these regions should be freed, and to which region each allocation site should write. Although very sophisticated, the Tofte/Talpin system relies critically on the fact that regions, region allocation, and region deallocation are introduced by the compiler and not by the programmer. Besides being fully automatic, the Tofte/Talpin system has the advantage that the runtime overhead for memory management is reduced to an absolute minimum while also being safe. Unfortunately, region inference is not perfect. To avoid leaking a great deal of memory it is necessary for the programmer to understand the regions inferred by the compiler and to adjust the program so that the compiler infers better region assignments. Second, optimizations beyond the basic inference procedure make an enormous difference in memory management performance [AFL95, BTV96]. Both of these properties suggest that explicit first-class regions may be appropriate, but combining explicit programmer-controlled regions with region inference appears to be a very difficult problem.

Bobrow [Bob80] is the first to propose the use of regions to make reference counting tolerant of cycles. This idea is taken up by Ichisugi and Yonezawa in [IY90] for use in distributed systems. Neither of these papers considers the use of regions for enhancing locality, nor do they include any performance measurements.

Grunwald and Zorn [GZ93] and Detlefs, Dosser and Zorn [DDZ94] study the performance of various allocators. Vo's paper on regions [Vo96] also compares the performance of the malloc/free-like allocator of the Vmalloc package with other malloc/free implementations. Grunwald, Zorn and Henderson compare the performance and cache locality of different allocators in [GZH93]. None of these studies consider region-based allocation.

## 3 Programming Model

We have implemented a prototype safe region-based memory management system as an extension of C, *C@*. Using C allows us to compare existing allocation-intensive programs with versions of these applications modified to use regions. Our prototype requires language and compiler support for two reasons. First, it is much easier for the compiler to generate the reference counting code than to insert reference counts by hand. Second, our implementation must locate all live local variables containing pointers; this information is only available in the compiler. An added advantage of modifying the language is that we can enforce some of the requirements on pointers to objects in regions at compile-time. The rest of this section presents C@, the region allocation library, and gives a simple example.

### 3.1 The Language C@

C@ distinguishes two kinds of pointers: normal pointers and *region pointers*, i.e., pointers to objects in regions. Region pointers are defined with '@' instead of '*,' e.g., `int @x`. The types $T$@ and $T*$ are different types, and no implicit conversion exists between them although explicit casts are allowed. These casts are unsafe, but are necessary for our experiments because the standard C libraries are not aware of region pointers. In particular `deleteregion` does not account for region pointers cast to normal pointers.

When a region pointer $r$ is updated it is potentially necessary to adjust two reference counts, one for the old region $r$ points to (a decrement) and one for the new region $r$ points to (an increment). This implies that region pointers must always be initialized, which is enforced by requiring initialization of all local variables that are, or contain, region pointers, and by clearing (writing 0's in) all objects allocated in regions. Because C@ must recognise all writes of region pointers, copying `structs` containing region pointers is forbidden—C's `unions` make it impossible to know which parts of a structure actually contain region pointers. Region pointers behave otherwise like normal ANSI C pointers and in particular address arithmetic is allowed.

While these restrictions on regions pointers in C@ are not onerous, it is worth noting that explicit regions could be integrated into more modern languages with fewer modifications. For example, in Java [GJS96] pointers (i.e., references) are always initialized and `union` does not exist.

```
typedef struct region @Region;
typedef size_t
  (*cleanup_t)(/* struct ??? @x */);
typedef size_t
  (*cleanuparray_t)(/*size_t n,
                      struct ??? @x */);

Region newregion(void);
int deleteregion(Region *r);

void @ralloc(Region r, size_t size,
             cleanup_t cleanup);
void @rarrayalloc(Region r,
                  size_t n, size_t size,
                  cleanuparray_t cleanup);
void @rstralloc(Region r, size_t size);

Region regionof(void @x);
```

Figure 2: Region allocation interface.

### 3.2 The Region Library

Figure 2 shows the region interface. A new region is created with `newregion`. Objects are allocated with `ralloc`, arrays with `rarrayalloc`. Objects or arrays that do not contain any region pointers can be allocated with `rstralloc`; the `cleanup` arguments to `ralloc` and `rarrayalloc` are discussed in Section 4. The memory returned by `ralloc` and `rarrayalloc`, but not `rstralloc`, is cleared. An object's region is returned by `regionof`.

An attempt to delete a region is made by calling `deleteregion(x)`. The deletion succeeds if there are no references (excepting `*x`) to the region in live variables or in other regions. On success, `*x` is set to `NULL`, and `1` is returned. On failure `*x` is unchanged, and `0` is returned.

Figure 3 shows a simple example that copies a list into a region, then later deletes that region. The `cleanup_list` function is presented in Section 4.

```
struct list {
  int i;
  struct list @next;
};

struct list @cons(Region r, int x,
                  struct list @l)
{
  struct list @p =
    ralloc(r, sizeof(struct list),
           cleanup_list);
  p->i = x; p->next = l;
  return p;
}

struct list @copy_list(Region r,
                       struct list @l)
{
  if (l == NULL) return NULL;
  else return cons(r, l->i,
                   copy_list(r, l->next));
}

void work(struct list @l)
{
  Region tmp = newregion();

  l = copy_list(tmp, l);
  ... do something with l ...
  deleteregion(&tmp);
}
```

Figure 3: List copy using regions.

## 4 Implementation

Our implementation of safe regions is based on the lcc [FH95] C compiler and a runtime library implementing our region interface. The modified lcc handles the language extensions of Section 3 and cooperates with the runtime library to maintain region reference counts. The target machine is a Sun UltraSparc-I. This section discusses how our implementation manages regions (Section 4.1) and reference counts (Section 4.2). We also have a modified version of our implementation that supports unsafe regions: it is identical to the safe version, except that all support for maintaining reference counts is disabled. The section concludes with a argument that the overhead of memory management is amortized constant time per instruction executed.

### 4.1 Managing Regions

The goal of the region library is to provide cheap object allocation and region deletion. It must also maintain a mapping from memory addresses to regions for the reference counting code.

The region data structure is shown in Figure 4. A region contains a reference count and two *allocators*, one for normal allocations (`ralloc` and `rarrayalloc`) and one for region-pointer-free data (`rstralloc`). Each al-

```
struct allocator {
  char *firstpage;
  /* offset at which to allocate */
  int allocfrom;
};

struct region {
  int rc;
  struct allocator normal;
  struct allocator string;
};
```

Figure 4: Region structure.

locator maintains a list of 4K byte pages, with allocation occurring only on the first page of the list.[1]

Allocation is very simple: If the allocation fits on the first page just return `firstpage+allocfrom` and increment `allocfrom`, if not allocate a new page and try again. The `ralloc` and `rarrayalloc` functions must also save the `cleanup` function at the start of the allocated object (see below) and clear the rest of the allocated memory. Finally, `rarrayalloc` must save the size of the array. The allocators maintain an array mapping page addresses (i.e., memory addresses / 4K) to regions.

The space overheads of this scheme are low: eight bytes per page for the map of pages to regions and the list of allocated pages. If an object does not fit in the space remaining at the end of a page that space is wasted.[2] Each allocation needs zero (for strings) to twelve (for arrays) bytes of bookkeeping information.

The region itself is stored in the first page allocated for that region. To reduce cache conflicts between region structures, successive regions are offset by 64 bytes (the 2nd level cache line size) in their first page, up to a maximum offset of 512.

### 4.2 Managing Reference Counts

While an object is allocated and deallocated only once, references to an object may change an arbitrary number of times. Thus, while object allocation and region deallocation are inexpensive in our system, maintaining reference counts is potentially very expensive. It is useful to distinguish between references in local variables, which change frequently, from references in the heap, which are updated more rarely. The main aim of our reference counting scheme is to avoid the large overhead that would be incurred by reference counting local variables exactly. There are four components to our scheme: maintaining approximate reference counts for local variables, maintaining exact reference counts for pointers in the heap, performing a scan of the stack, and scanning deleted regions.

[1] Our prototype only handles allocations of less than one page—our benchmarks did not use any larger objects. This restriction could be lifted without affecting the cost of small allocations.

[2] Also the last byte of a page cannot be used if pointers to the end of objects are supported.

### 4.2.1 Local Variables

The *exact reference count* for a region is the number of pointers to objects of that region from other regions, global storage (including global and static variables and any memory returned by `malloc` if used) and the live variables in all active call frames. We need the exact reference count only when `deleteregion` is called; at all other times we need only maintain enough information to compute this reference count (this is the principle behind *deferred reference counting* [DB76]).

The *actual reference count* stored with a region reflects the number of pointers to objects of that region from all other regions, global storage, and the live variables in all active call frames above the *high water mark* on the stack (note the stack grows downward on the SPARC). The high water mark is just a location on the stack, with some frames above and some below. Our system maintains the following invariant:

> ($*$) *The number of frames below the high-water mark is always at least one.*

Thus writes to local variables never update reference counts.

Invariant ($*$) is maintained by a procedure call, but some work may be required on procedure return. If control returns to a call frame at the high-water mark, then the region reference counts attributable to local variables are decremented and the high water mark is adjusted above the call frame. We describe this `unscan` function in further detail below.

When `deleteregion(r)` requires the exact reference count of $r$ it scans the portion of the stack below the high water mark and updates the region reference counts. At that point the actual and exact reference counts are equal. The stack scan sets the high water mark above the frame of `deleteregion`, which is not itself scanned.

### 4.2.2 Global and Region References

The compiler generates code to update reference counts on writes of region pointers in global storage and to objects within regions. Reference counting is different for writes to global storage and for writes within regions. When a pointer to region $r$ is written in region $r$, the reference count for $r$ is not incremented. We call such pointers *sameregion* pointers. As global storage does not belong to any region, it cannot contain sameregion pointers. Figure 5 shows pseudo-code for both kinds of reference count updates. The instruction counts reflect the number of SPARC instructions required by our implementation for each kind of write.

Our compiler attempts to distinguish writes to local variables, global storage and regions at compile-time, but this is not always possible in C because writes via normal pointers can be writes to global storage or to variables on the stack. Writes to the stack should only be reference counted if that variable is above the high water mark. For writes that cannot be statically distinguished, a more expensive runtime routine is used to determine which case applies.

Global writes - 16 instructions

```
t = *a;
if (regionof(t) != regionof(b)) {
  regionof(t)->rc--;
  regionof(b)->rc++;
}
```

Region writes - 23 instructions

```
t = *a
if (regionof(t) != regionof(b)) {
  if (regionof(t) != regionof(a))
    regionof(t)->rc--;
  if (regionof(b) != regionof(a))
    regionof(b)->rc++;
}
```

Figure 5: Reference count methods for `*a=b`.

### 4.2.3 Stack Scan

To allow the stack to be scanned at runtime, the compiler records at each function call site the set of registers and offsets in the current call frame that contain live region pointers. Because our implementation is based on lcc, which does not have liveness information available, our prototype considers all variables in scope to be live. The liveness information is static data whose location is recorded in the unused bits of a NOP instruction at the call site. (A more complex implementation would avoid this extra instruction.) After the scan of a call frame increments the region reference counts for all live variables, the high water mark is placed just below that call frame.

A call frame that was scanned is unscanned automatically when control returns to that frame. This is achieved by modifying return addresses during the scan to point to a special `unscan` function that decrements the region reference counts, adjusts the high water mark above the call frame, and then jumps to the original return address.

### 4.2.4 Region Scan

A deleted region $r$ may contain pointers to objects in other regions. To adjust the reference counts of other regions we examine all the region pointers in objects allocated in $r$. The user supplies the function that performs this task as the `cleanup` argument to `ralloc` and `rarrayalloc`. This function must call `destroy` on every region pointer in the allocated object and return the size of the object. We require the user to provide this function for the same reason that we forbid copying structures that contain region pointers: the presence of C's `unions` makes it impossible for the compiler to locate every region pointer. For cases without `union`, and in higher-level languages, the `cleanup` function could be generated automatically by the compiler. The `cleanup` function also allows object finalization. Figure 6 shows the `cleanup_list` function for the `list` type of Figure 3. The pseudo-code for scanning deleted regions is in Figure 7.

```
size_t cleanup_list(struct list @x)
{
  destroy(x->next);
  return sizeof *x;
}
```

Figure 6: Example of `cleanup` function.

```
for all pages p of region:
  deleting = p;
  end = p + PAGESIZE;
  while (deleting < end)
    cleanup_t cln = *(cleanup_t *)deleting;
    /* the end of unfilled pages is marked
       with a NULL */
    if (!cln) break;
    deleting += sizeof(cleanup_t);
    s = cln(deleting);
    deleting += ALIGN(s, ALIGNMENT);
```

Figure 7: Region cleanup.

### 4.3 Amortized Cost of Safe Regions

The primary justification for our reference counting scheme is that it both solves the engineering problem of avoiding maintaining all reference counts all the time while still bounding the cost of memory management. We make two assumptions. First, we assume that the size of the largest stack frame is bounded by a constant $c$.[3] Second, we assume that every allocated word is actually referenced by some program instruction. We can then argue that the amortized cost of memory management is a constant per instruction executed.

The costs in our region system are incurred on region allocation, object allocation, updates (reference counting), region deletion (scanning regions and the stack), and on procedure return (unscanning). To show our bound we distribute these costs over all memory references and procedure calls in such a way that the maximum cost associated with any operation is bounded by a constant.

Allocating a new region is a constant time operation. The cost of allocating an object $o$ is at worst proportional to the size of $o$ plus the cost of acquiring a new page. The latter cost is constant, the former is distributed to the program references to $o$. Scanning a deleted region simply scans each object $o$ in the region, the cost of which can also be assigned to program references to $o$. A reference count operation is charged to the assignment.

The analysis of the stack scan and unscan requires a bit of care. Every scan of a frame is eventually paired with a corresponding unscan of the same frame. Except for the frame for `deleteregion` (which is at the bottom of the downward-growing stack), the cost of scanning and unscanning a frame $f$ is charged to the call that creates the frame immediately below $f$ on the stack. Because the high-water mark moves up only on procedure return, it follows that every function call is charged for the scan and unscan of at most one frame. Assuming stack frames have at most size $c$, every call is therefore charged a constant cost.

[3] Actually, it is sufficient if the number of live region pointers in a frame is bounded by a constant.

## 5 Results

We describe our benchmarks and the changes required to adapt them to use regions in Section 5.1. We use these benchmarks to compare the performance of regions with three malloc/free implementations (described in Section 5.2) and the Boehm-Weiser conservative garbage collector. We find that the safe region-based programs use from 9% less to 19% more memory (Section 5.4) than the allocator that uses least memory, and are from 5% slower to 9% faster than the fastest allocator. Unsafe regions are never slower than other allocators and are up to 16% faster (Section 5.5). On one benchmark, we use regions to group frequently accessed data structures and obtain a 24% performance improvement. We measure the overhead of safe regions and find that it does not exceed 17% and is generally much lower (Section 5.6).

### 5.1 Benchmarks

We compared the performance of our safe, region-based memory management on six allocation-intensive C programs. These programs and the inputs we used for our measurements are

- *cfrac*: A program to factor large integers using the continued fraction method. The original application uses explicit reference counting to reclaim storage. We factor 4175764634412486014593803028771.
- *gröbner*: Find the Gröbner's basis of a set of polynomials. The input is nine nine-variable polynomials.
- *mudlle*: A byte-code compiler for a scheme-like language. The original version of this program uses unsafe regions. The same 500-line file is compiled 100 times.
- *lcc*: Our modified version of the lcc C compiler. The original program also uses unsafe regions (Hanson's arenas [Han90]). The input is a 6000-line C file.
- *tile*: Automatically partitions a set of text files into subsections based on frequency and grouping of words in the text. This program uses malloc/free. Twenty copies of a 14K text are given as input.
- *moss*: A software plagiarism detection system, written originally using malloc/free. The input is 180 student compiler projects (about 10MB).

We modified these programs to use our safe regions. Our first step was to choose appropriate regions for those applications that were not already region-based. All the applications have a simple region structure, even when the data structures stored in the regions are very complex. For instance, our region-based 'cfrac' creates a region for temporary computations for every few iterations of the main algorithm. Partial solutions are copied from this region to a solution region so that old temporary regions can be deleted. In 'mudlle,' one region holds the abstract syntax tree of the file being compiled and one region is created to hold the data structures needed to compile each function. The other programs have similarly simple region structures. In general, we found it fairly easy to modify these benchmarks to use regions. The difficulty lay not so much in selecting where to create and delete regions, but in the tedious process of changing types, writing cleanup functions, etc. As pointed out above, most of this work would not be necessary in a higher-level language. The other difficulty is finding stale pointers that prevent a region from being deleted; an environment for debugging regions would be helpful here.

Once the region structure is selected, the following basic modifications are made: calls to `malloc` and `free` are replaced with appropriate region operations, normal pointers are changed to region pointers, the `cleanup` functions are written and initializations are added for all local region pointers. Each application has some further changes:

- For *cfrac* we disable the explicit reference counting and allocate some static objects in regions. We also add the copies of partial solutions to the solution region.
- For *mudlle* it is necessary to clear some global variables with stale pointers in the original code; otherwise these pointers prevent region deletion.
- In *gröbner* we must replace some bulk copies (via assignment) with explicit copies, statically allocate some structures that were originally on the stack, and add copies of the polynomials that form the basis to a result region. Many frees are replaced by clearing the corresponding pointer, a number of other pointers must also be explicitly cleared.
- For *lcc* we replace bulk copies (via assignment or `memcpy`) with calls to hand-written copy functions and replace some uses of `memset` with explicit `NULL` writes. Information is added to some types so that `cleanup` functions can be written. Some static and stack objects are allocated in regions and some global variables and region-allocated objects are cleared. Memory for strings is allocated individually rather than in blocks. To improve 'lcc's performance, we create a region for every hundred statements compiled rather than for every statement.
- For *tile*, one local variable must be cleared to allow a region to be deleted. In addition, numerous memory management bugs present in the original code are repaired.
- *moss* allocates some large static arrays in a region.

| Name | Lines | Changed lines |
|---|---|---|
| cfrac | 4203 | 149 / 18 |
| gröbner | 3219 | 282 / 111 |
| mudlle | 5078 | 402 / 22 |
| lcc | 12430 | 1223 / 349 |
| tile | 926 | 159 / 10 |
| moss | 2675 | 88 / 4 |

Table 1: Complexity of benchmark changes.

The size of these changes is summarized in Table 1. The 'Lines' columns counts the number of lines in the original source code. The first number under 'Changed lines' represents the number of changed or extra lines of code in the region-based version, based on the results of `diff -f`; the second number counts only those lines that are not part of the basic modifications.

### 5.2 Allocators

We compare the performance of regions with the following allocators:

- *Sun*: This is the default allocator supplied with Solaris 2.5.1. It provides an interesting point for comparison as it will likely be used by default.
- *BSD*: The version of the BSD memory allocator supplied by Sun. It rounds allocations up to the nearest power of two. It features fast allocation and deallocation but has a very large memory overhead.
- *Lea*: Doug Lea's implementation of `malloc`, v2.6.4.[4] This is an improved version of the allocator used in some previous surveys of memory allocation costs [DDZ94, Vo96]. In those surveys this allocator exhibited good performance overall.
- *GC*: The Boehm-Weiser conservative garbage collector [BW88] v4.12. We disable all `free`'s when compiling with this collector, thus guaranteeing safe memory management.

We use three different region libraries in these measurements:

- *safe*: the safe region-based memory management described in Section 4. This library is used for the region-based measurements (the 'Reg' bars in Figures 8, 9 and 10).
- *unsafe*: the same as *safe*, but with all operations that maintain or test reference counts disabled. This library is used for the unsafe region measurements in Figure 9 (bar 'unsafe').
- *emulation*: a region library that uses malloc and free to allocate and free each individual object. This library approximates the performance a region-based application would have if it were written with malloc/free. In our experiments this

[4] Available at ftp://g.oswego.edu/pub/misc/malloc.c

| Name | Total allocs | Total kbytes allocated | Max. kbytes allocated | Total regions | Max. regions | Max. kbytes in region | Avg. kbytes per region | Avg allocs per region |
|---|---|---|---|---|---|---|---|---|
| cfrac | 3812425 | 60107 | 106 | 23383 | 5 | 83.6 | 2.57 | 163 |
| gröbner | 805321 | 28454 | 43.6 | 11452 | 4 | 13.0 | 2.48 | 70 |
| mudlle | 737850 | 10661 | 240 | 4648 | 13 | 141 | 2.29 | 159 |
| lcc | 177816 | 8711 | 4567 | 1249 | 3 | 4125 | 6.97 | 142 |
| tile | 40699 | 1347 | 88.4 | 81 | 5 | 41.9 | 12.5 | 502 |
| moss | 552240 | 7778 | 2212 | 1899 | 7 | 1246 | 3.49 | 291 |

Table 2: Allocation behaviour with regions.

| Name | Total allocs | Total kbytes allocated | Max. kbytes allocated |
|---|---|---|---|
| cfrac | 3809160 | 66879 | 84.8 |
| gröbner | 804956 | 28449 | 46.2 |
| mudlle | 742495 | 13578 | 324 |
| (w/o overhead) | | 10678 | 239 |
| lcc | 166495 | 9102 | 4683 |
| (w/o overhead) | | 8452 | 4375 |
| tile | 40678 | 1330 | 84.0 |
| moss | 552240 | 7778 | 2203 |

Table 3: Allocation behaviour with `malloc`.

library is used to measure the performance of the 'mudlle' and 'lcc' with malloc/free allocators.

Using this library imposes a small space overhead: the objects allocated in a region must be kept in a linked list so they can be freed when `deleteregion` is called. Table 3 and Figure 8 include additional entries estimating memory usage without this overhead.

The C library sometimes calls `malloc`, and thus C applications using regions inevitably include a mix of memory allocated with regions and with `malloc`. The region-based programs are linked with the default 'Sun' allocator; our results for regions include the time and space cost of these calls to `malloc`.

### 5.3 Allocation Characteristics

Tables 2 and 3 give the memory allocation characteristics of the region-based and malloc/free versions of the applications. 'Total allocs' is the total number of memory allocations performed by the program and 'Total kbytes allocated' is the total number of kilobytes allocated, with allocation sizes rounded to the nearest multiple of four. The 'Max. kbytes allocated' column contains the maximum amount of memory allocated at any time. The remaining columns concern only regions: 'Total regions' is the number of regions created, 'Max. regions' is the maximum number of regions present at any time, 'Max. kbytes in region' is the size of the application's largest region, 'Avg. kbytes per region' is the average size of the regions and 'Avg. allocs per region' is the average number of objects allocated per region.

The discrepancies in Tables 3 and 2 in the number of allocations and the amount of memory allocated are generally small and attributable to the small changes needed to convert the applications to use regions. The first exception is 'cfrac', where the region-based version does not need to allocate space for reference counts, but does need to allocate some extra copies of the results. The second exception is 'lcc' where the region-based version does more than 10,000 extra allocations because strings are allocated individually and some stack allocated structures are converted to region allocated structures.

Because a region can only be deleted all at once, the region-based versions of the applications tend to free memory later than the malloc/free-based versions. Thus the maximum amount of memory allocated at any time tends to be slightly larger in Table 2 than in Table 3.

### 5.4 Memory Usage

Figure 8 compares the amount of memory requested from the operating system (bar 'OS') by the different allocators with the memory actually requested by the programmer (bar 'requested,' see Tables 3 and 2). The graphs for 'cfrac' and 'tile' are clipped: the Boehm-Weiser garbage collector used 832 and 664 kbytes respectively. For 'lcc' and 'mudlle,' the first bar is the raw memory usage, the second bar has the region-emulation overhead removed. The 'cfrac'/Boehm-Weiser garbage collector pair requests less memory than the malloc/free-based versions because it does not need reference counts.

Regions use from from 9% less to 19% more memory than Doug Lea's allocator. Regions use less memory than all other allocators in all other cases, except on the 'tile' benchmark where regions use 1% more than Sun's allocator. The BSD allocator and the Boehm-Weiser garbage collector use a lot of memory, which makes them unsuitable for some applications.

### 5.5 Performance

For each application/allocator combination we measure wall-clock execution time ('base+memory' in Figure 9), including the portion of time spent in memory management ('memory' only). Figure 10 reports processor cycles lost to read (waiting for the result of a load instruction) and write (store buffer full) stalls. An allocator that uses the memory hierarchy more efficiently loses fewer cycles to read and write stalls. All measurements

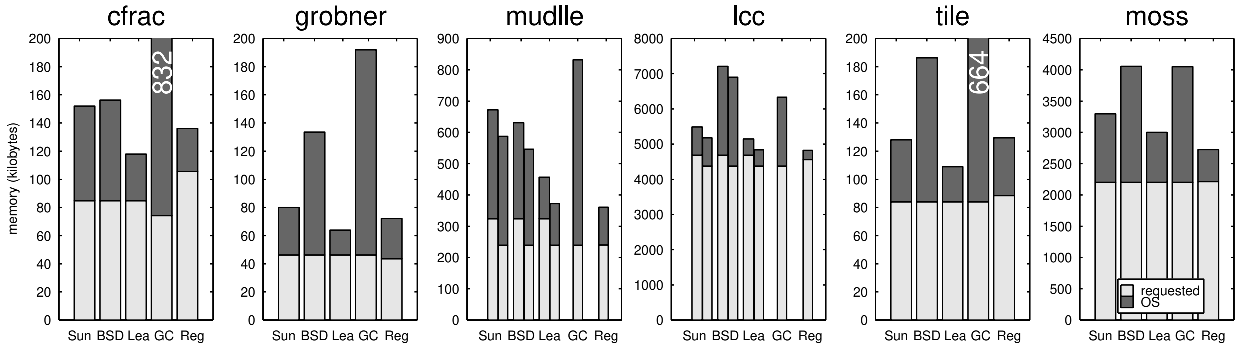


Figure 8: Memory overhead.

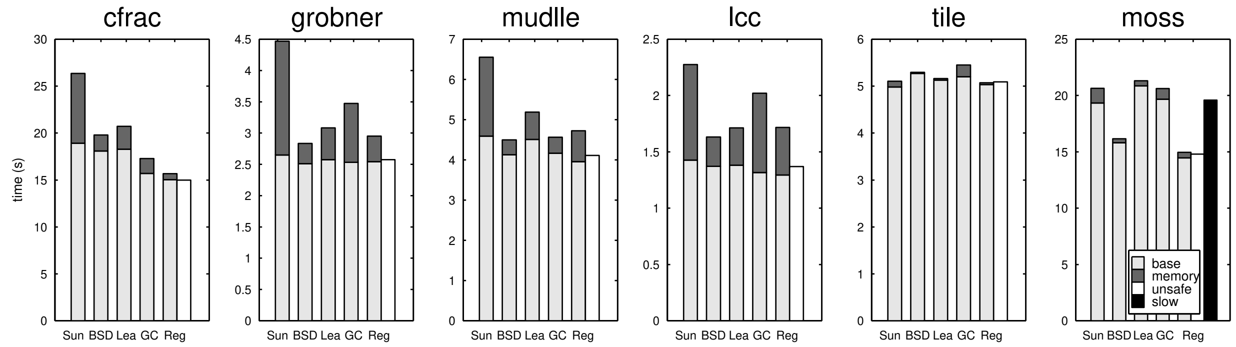


Figure 9: Execution time and memory management overhead.

are performed on 167Mhz UltraSparc-I workstation and use the UltraSparc's internal counters for precision.

All applications are compiled with lcc, a non-optimizing compiler. The time spent in the actual application is represented by the 'base' part of the execution time in Figure 9. The allocation libraries are compiled with optimization by the GNU C compiler or are supplied by Sun. The portion of time spent in these libraries (and in reference counting for region-based allocation) is the 'memory' part of the execution time, again in Figure 9. Compiling the applications with an optimizing compiler will not change the time spent in memory management and will reduce the 'base' part of execution time uniformly for all allocators. Thus using an optimizing compiler would not change the results of our comparison.

On these benchmarks, unsafe regions (bar 'unsafe' in Figure 9) are faster (up to 16%) than all the other allocators. Safe regions are as fast or faster (up to 9%) than the other allocators on 'cfrac', 'tile', and 'moss' and only slightly slower than the BSD allocator (5% slower) and Boehm-Weiser garbage collector (3% slower) on 'mudlle'. On 'lcc' safe regions are slower than the BSD allocator (5% slower) and competitive with Doug Lea's allocator—this application has the highest overhead for safe regions.

The graph for 'moss' in Figure 9 includes the time for an optimised version ('base+memory' bar) and our original region version ('slow' bar). The memory allocation pattern of 'moss' is to alternately allocate a small, frequently accessed object and a large, infrequently accessed object. This pattern reduces memory locality among the small objects. The 24% improvement in execution time in 'moss' is obtained by using two regions: one for the small objects and one for the large objects. This improvement is also reflected in Figure 10: the graph for 'moss' shows that the optimized region version ('Reg' column) has approximately half the stalls of the original version ('Slow' column). It is interesting to note that the BSD memory allocator (which automatically segregates objects by size) tends to have fewer stalls than the other explicit allocators; the resulting performance advantage is most visible with 'moss.'

### 5.6 Cost of Safety

The costs for safe regions can be divided into three parts that mirror the implementation: the cost of calling the `cleanup` functions when regions are deleted, the cost of scanning the stack when `deleteregion` is called, and the cost of maintaining the reference counts on region pointer writes. Figure 11 gives the breakdown of these costs for our six applications.

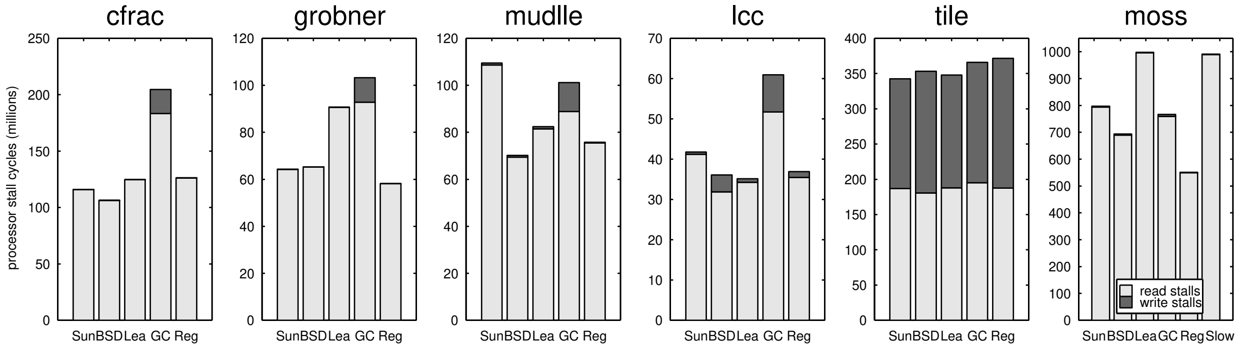


Figure 10: Processor cycles lost to stalls.

The cost of safety varies from negligible ('tile') to 17% ('lcc'). For less allocation and pointer intensive programs, we expect results similar to 'tile.' We have considered various methods of reducing the cost of safety, such as recognizing sameregion pointers at compile-time, and various schemes for optimizing the `cleanup` of regions. We plan to implement some of these ideas in another version of region-based memory management.

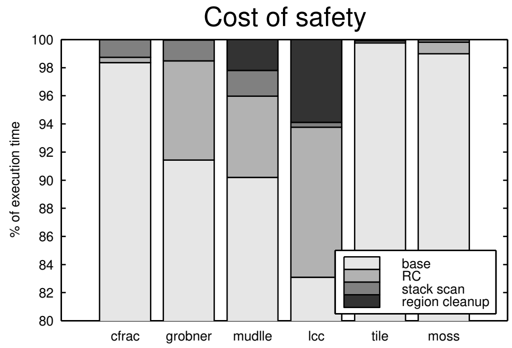


Figure 11: Region costs.

## 6 Conclusion

We have presented a new region-based memory management technique that combines efficiency with safety. We have shown that this technique often uses less memory and is as fast or faster than traditional malloc/free-based memory management. Safe regions are also faster than conservative garbage collection in most cases and use much less memory. The programmer can use regions to explicitly take advantage of the locality of dynamically allocated data structures. This can lead to much better performance, as the 'moss' example shows.

Our style of region-based memory management requires extensions to be useful for all applications. We plan to address this issue as part of providing region-based memory management in Titanium [YSP+98], an explicitly-parallel, Java-based [GJS96] programming language.

## References


[AFL95] Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, La Jolla, CA, June 1995.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[Bob80] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 187–196, Albuquerque, New Mexico, June 1993.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6), 1994.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[GZ93] Dirk Grunwald and Benjamin Zorn. Customalloc: Efficient, synthesised memory allocators. *Software Practice and Experience*, 23:851–869, 1993.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 177–186, Albuquerque, New Mexico, June 1993.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[IY90] Yuuji Ichisugi and Akinori Yonezawa. Distributed garbage collection using group reference counting. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

[Ros67] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[SO96] David Stoutamire and Stephen Omohundro. The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, August 1996.

[Sto97] D. Stoutamire. *Portable, Modular Expression of Locality.* PhD thesis, University of California at Berkeley, 1997.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[Vo96] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 26(3):357–374, March 1996.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, September 1992. Springer-Verlag.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 1–14, Palo Alto, CA, February 1998.