Safe Manual Memory Management in Cyclone

Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman and Trevor Jim

Abstract

The goal of the Cyclone project is to investigate how to make a low-level C-like language safe. Our most difficult challenge has been providing programmers control over memory management while retaining safety. This paper [1] describes our experience trying to integrate and use effectively two previously-proposed, safe memory-management mechanisms: statically-scoped regions and tracked pointers. We found that these typing mechanisms can be combined to build alternative memory-management abstractions, such as reference counted objects and arenas with dynamic lifetimes, and thus provide a flexible basis. Our experience—porting C programs and device drivers, and building new applications for resource-constrained systems—confirms that experts can use these features to improve memory footprint and sometimes to improve throughput when used instead of, or in combination with, conservative garbage collection.

1 Introduction

Low-level languages such as C provide a degree of control over space, time, and predictability that high-level languages such as Java do not. But the lack of safety for C has led to many failures and security problems. The goal of our research is try to bring the "Mohammad of safety" to the "mountain of existing C code."

[1] This work is based on an earlier work [1]: Experience with safe manual memory-management in cyclone, in ISMM '04: Proceedings of the 4th international symposium on Memory management, (c) ACM, 2004. http://doi.acm.org/10.1145/1029873.1029883 This paper improves on the framework presented in the conference version by strictly separating pointer tracking and regions; this permits unique or reference-counted pointers to any region, and enables supporting reaps, which we discuss in some detail. The exposition has been generally revised and improved. All experimental results reported in [1] have been revised to use the new framework, and we additionally present experience with porting three Linux device drivers and one scientific application to Cyclone.

Email addresses: nswamy@cs.umd.edu (Nikhil Swamy), mwh@cs.umd.edu (Michael Hicks), greg@eecs.harvard.edu (Greg Morrisett), djg@cs.washington.edu (Dan Grossman), trevor@research.att.com (Trevor Jim).

Toward that end, we have been developing Cyclone [2], a safe dialect of C. Cyclone uses a combination of programmer-supplied annotations, an advanced type system, a flow analysis, and run-time checks to ensure that programs are safe. At first, we relied entirely on heap allocation and the Boehm-Demers-Weiser (BDW) conservative garbage collector to reclaim memory. BDW [3] interoperates with legacy libraries and supports polymorphism without needing run-time type tags.

The BDW collector is convenient, but it does not always provide the performance or control needed by low-level systems applications. In previous work [4], we described an integration of BDW with safe stack allocation and LIFO arena allocation; in both cases all objects are deallocated at once when the relevant scope is exited. A region-based type-and-effect system based upon the work of Tofte and Talpin [5] ensured safety while providing enough polymorphism for reusable code to operate over data allocated anywhere. In practice, we found that supporting stack allocation was crucial for good performance, and our system was able to infer most region annotations for porting legacy C code that used stack allocation. We found that LIFO arenas are useful when callers know object lifetimes but only callees can determine object sizes. Unfortunately, LIFO arenas suffer from several well-known limitations that we encountered repeatedly. In particular, they are not suited to computations such as event loops in server programs.

Since then, we have explored the integration of *tracked pointers* into our memory-management framework. Our tracked pointers are closely related to typing mechanisms suggested by others, including linear types [6], ownership types [7], alias types [8], and capability types [9]. The critical idea in these proposals is to make it easy to track the state of an object locally by restricting aliasing. For example, Cyclone's tracked pointers include *unique pointers*. A value with a unique-pointer type is guaranteed to be the only (usable) reference to an object. Such objects can be explicitly deallocated at any program point, and a modular flow analysis ensures that the dangling pointer is not subsequently dereferenced.

Tracked pointers are not a novel idea, but we found many challenges to implementing them in a full-scale safe language, where they must be integrated with other features such as exceptions, garbage collection, type abstraction, the address-of operator, undefined evaluation order, etc. To our knowledge, no one has attempted to address all these features in a full-scale language implementation.

We found great synergy between tracked pointers and regions. In particular, we use the LIFO region machinery to support a form of "borrowed" pointers [10], which goes a long way in easing the burdens of tracking. We also use unique pointers as capabilities for building further memory-management abstractions. In particular, unique pointers control access to a form of dynamically-scoped arenas [11], and a slight extension supports reference-counted objects. Finally, tracked pointers naturally combine with regions to support safe *reaps* in which objects can either be deallocated individually or freed all at once; reaps can be useful for tuning the

inherent space-time tradeoff between individual and bulk deallocation [12].

In this paper, we describe our support for tracked pointers and the extensions they enable; the Cyclone manual has further detail. [2] We then discuss our experience using these facilities to build or port a few target applications, including a multimedia overlay network, a web server, a Scheme interpreter, an ftp server, an image-manipulation program, two numerical-analysis applications and three Linux device drivers. These applications were chosen for one of three reasons : they are structured as (infinite) loops with loop-carried state and are thus not well-suited for LIFO arenas; they might be used as subroutines on resource-limited platforms, such as cell-phones or other embedded systems; they are used in settings in which garbage collection is infeasible or undesirable. In most of these applications, we were able to reduce, if not eliminate, the need for garbage collection. We also saw dramatic improvements in working-set size, and, for one application, an improvement in throughput.

Thus, the contributions of this paper are two-fold:

(1) We show that the addition of tracked pointers to a region-based language provides a flexible basis for safe manual memory management, which can complement or replace garbage collection.
(2) We confirm that the resource requirements of some important applications can be significantly improved through manual memory management and that Cyclone's safety restrictions do not prevent this improvement.

2 Regions in Cyclone

A *region* is a logical container for objects that obey some memory-management discipline. For instance, a stack frame is a region that holds the values of the variables declared in a lexical block, and the frame is deallocated when control-flow exits the block. As another example, the garbage-collected heap is a region, whose objects are individually deallocated by the collector.

In Cyclone, each region is given a compile-time name, either explicitly by the programmer or implicitly by the compiler. For example, the name of the heap region is `‘H`, and the region name for the stack frame of a function `foo` is `‘foo`. If `‘r` is the name of a region, and an object o with type `T` is allocated in `‘r`, then the type of a pointer to o is written `T* @region(‘r)`, or `T*‘r` for short. To ensure that programs never dereference a pointer into a deallocated region, the compiler tracks a conservative approximation of (a) the regions into which a pointer can point, and (b) the set of regions that are still live at each program point. This is implemented

[2] `http://www.eecs.harvard.edu/~greg/cyclone/`

```
FILE *infile = ...
if ( get_tag(infile) == HUFFMAN_TAG ) {
  region<`r> h;
  struct code_node<`r> *`r huffman_tree;
  huffman_tree = read_huffman_tree(h, infile);
  read_and_decode(infile, huffman_tree, symbol_stream, ...);
  /* region freed here */
} else ...
```

Fig. 1. LIFO Arena example

using a type-and-effects system in the style of Tofte and Talpin [5].

Cyclone supports *region polymorphism*, which lets functions and data structures abstract over the regions of their arguments and fields. By default, Cyclone assumes that pointer arguments to functions point into arbitrary regions, but that all these regions are live (the assumption of liveness is enforced at each call-site). A unification-based algorithm infers instantiations for region-polymorphic functions and the regions for local variables. This drastically cuts the number of region annotations needed for programs (we include explicit annotations in our examples for clarity). Cyclone also supports region subtyping based on region lifetimes, which combines with region polymorphism to make for an extremely flexible system. In practice, we have found few (bug-free) examples where stack-allocation could not be easily ported from C to Cyclone.

2.1 LIFO Arenas

This basic region system easily supports a form of *arenas* that have stack-like last-in-first-out (LIFO) lifetimes, but also support dynamic allocation. [3] A LIFO arena is introduced with a lexically-scoped declaration:

```
{ region<`r> h; ... }
```

Here, `h` is a region *handle* having type `region_t<`r>` that can be used to allocate in the newly introduced region `` `r ``. At run-time, calling the primitive `rmalloc(h,...)` allocates space within region `` `r ``. We implement arenas as a list of pages acquired from the heap. Each call to `rmalloc` attempts to allocate memory within the current page (using pointer-bumping), or else allocates a new page. When the declaring lexical scope concludes all allocated pages are freed.

[3] A prior paper [4] referred to LIFO arenas as *dynamic regions* due to their dynamic sizes; in this paper we use the term *region* more generally, using *arena* to signify a region supporting dynamic allocation, and *LIFO* to signify scoped lifetime.

We have found that arenas work well for situations where data's lifetime is scoped, but the caller does not know how much space to pre-allocate on its stack frame. Consider the example in Figure 1 (adapted from the *Epic* benchmark in Section 4). If the image in `infile` is compressed using Huffman encoding, then the huffman tree is deserialized from the file by `read_huffman_tree` into a `code_node` tree allocated in region `` `r ``. This tree is used to decompress the remaining file contents into the `symbol_stream` array. The tree is not needed beyond the `if` block in which its region is defined, and is freed with that region when control exits the block. Obviously, statically allocating space for the tree would be problematic since the tree's size depends on the contents of `infile`.

Unfortunately, the LIFO restriction on arena lifetimes can limit their applicability. We often wanted to deallocate an arena before the end of its scope. This was particularly problematic for loops: If one pushes the arena declaration inside the loop then a fresh arena is created and destroyed for each iteration. Thus, no data in the arena can persist across iterations unless they are copied to an arena declared outside the loop. But then all data placed in an outer arena would persist until the loop terminates. For loops that do not terminate, such as a server request loop or event loop, the LIFO restriction can lead to unbounded storage requirements.

3 Tracked Pointers

Often the limitations of stack allocation and LIFO arenas can be conveniently overcome by using the heap region, whose contents are periodically garbage collected. However, garbage collection (GC) may not always meet an application's performance needs. For example, embedded systems, OS kernels, and network servers sometimes require bounds on space consumption, pause times, or throughput that may be hard to achieve with GC. Therefore, we extended Cyclone with a suite of mechanisms that would permit manual object deallocation without imposing a LIFO restriction. Our goal is not necessarily to eliminate GC, but rather to provide programmers with better control over tuning the space and time requirements of their programs.

In general, ensuring that manual deallocation is safe requires precise information regarding which pointers may alias other pointers. Though there are impressive analyses that compute such aliasing information [13], they usually require the whole program to achieve any level of accuracy. An alternative solution is to restrict or avoid aliasing altogether, so that reasoning about type-states can be done locally. One extreme is to require that objects which are deallocated be referred to by only a single alias-free pointer, i.e. the pointer is *unique*. In what follows, we describe how we incorporated unique pointers into Cyclone. Using basic support for tracking unique pointers, we have implemented safe per-object deallocation, reference counting, reaps [12], and arenas with non-scoped lifetimes. Tables 1 and 2

at the end of this Section (page 18) summarize Cyclone's memory-management constructs.

3.1 Basics

A pointer into any region can be designated a unique pointer. For now, we consider unique pointers into the heap; Section 3.4 considers unique pointers into other regions. A pointer's type is qualified by its *aliasability*: The `\A` aliasability designates aliasable (non-unique) pointers, whereas `\U` designates unique pointers. We write `T* @aqual(\U)` (or `T*\U` for short) to classify a unique pointer to an object of type `T`. For brevity, in this paper we assume that pointers without an explicit region annotation point to the heap (`‘H`); those without an explicit aliasability are freely aliasable (`\A`). A unique pointer into the heap is created by calling `malloc` and can be deallocated via `free`.

We use an intraprocedural, flow-sensitive, path-insensitive analysis to track when a unique pointer becomes *consumed*, in which case the analysis rejects a subsequent use of the pointer. We chose an intraprocedural analysis to ensure modular checking and a path-insensitive analysis to ensure scalability. To keep the analysis simple, a copy of a unique pointer in an assignment is treated as consuming the pointer. This ensures that there is at most one usable alias of a unique pointer at any program point. Here is an example:

```
int *\U q = p;  // consumes p
*q = 5;         // OK: q is not consumed here
free(q);        // consumes q
*p = *q;        // illegal: p and q are both consumed here
```

The first assignment aliases and consumes `p`, while the call to `free` consumes `q`. Therefore, the attempts to dereference `p` and `q` in the last statement are illegal. Dereferencing a (non-consumed) unique pointer does not consume it since it does not copy the pointer, as the first dereference of `q` shows. By default, we assume that functions do not consume their arguments. A function type augmented with an attribute `consume(i)` indicates that the i*th* parameter is consumed; the type of `free` has such an attribute. Our analysis rejects a function that consumes parameters unless the appropriate consume attributes appear in the function's type.

We allow unique pointers to be placed in containers that might have multiple aliases (e.g., a global variable or an aliasable object). This must be handled with care. Consider the example in Figure 2. The function `bar` allocates a unique pointer `p`, and then an aliasable pointer `pp` that points to `p`. [4] Within the function `bad`, pointers

[4] The type `int*\U*\A` is parsed `(int*\U)*\A`; i.e., an aliasable pointer to a unique pointer.

```
int bad(int *\U*\A x, int *\U*\A y) {
  free(*x);
  **y = 1; // dangling pointer dereference!
}
int bar() {
  int *\U p = malloc(sizeof(int));
  int *\U*\A pp = malloc(sizeof(int *));
  *p = 1;
  *pp = p;
  bad(pp,pp);
}
```

Fig. 2. Pitfall of accessing unique pointers in shared objects

x and y are aliases, so that p's storage is freed in the first statement via x, and then incorrectly accessed in the second statement via y. Cyclone rejects this program, as described below.

In many systems, reading a unique pointer is treated as a *destructive* operation that overwrites the original copy with NULL, so as to preserve the uniqueness invariant. This fixes the above example because reading *x would store NULL, so the access through y would result in a (safe) null pointer exception. Cyclone has pointer types that do not admit NULL as a value, so destructive reads are not permissible in general. Therefore, we provide an explicit *swap* operation (":=:") that allows one to swap one unique object for another (including NULL where permitted). We require the use of swaps whenever a field that holds a unique pointer value is to be read from within a shared structure, irrespective of whether or not the field admits the NULL value. Though notationally less convenient than a destructive read, we found that programming with swaps made us think harder about where NULL-checks were needed, and helped eliminate potential run-time exceptions. Using swap, we can correct the body of bad to be

```
int *\U xp = NULL;
(*x) :=: xp;
free(xp);
**y = 1; // NULL pointer exception
```

Reads to unique pointers must always be via a *unique path* u, having the form

$$u ::= \texttt{x} \mid u.\texttt{m} \mid u\texttt{->m} \mid *u$$

where x is a local variable [5] and u is a unique pointer. With this restriction, it is easy to verify that at any program point, there is at most one usable copy of any unique value. Furthermore, by making swap atomic, this property holds even if multiple threads were to execute init concurrently.

[5] We also require that the address of x has not been taken.

3.2 Borrowing Unique Pointers

Unique pointers make it easy to support explicit deallocation, but often force awkward coding idioms to maintain uniqueness. For example, we forbid pointer arithmetic on unique pointers, since doing so could let the user call `free` with a pointer into the middle, rather than the front, of an object, confusing the allocator. [6] As another example, we often want to pass a unique pointer to a library function without consuming it, even though the function may create benign, temporary aliases.

Most systems based on uniqueness or ownership have some way of creating "borrowed" pointers to code around these problems. A borrowed pointer is a second-class copy of a unique pointer that cannot be deallocated and cannot escape. This ensures that if we deallocate the original pointer, we can invalidate all the borrowed copies, or else we can prevent deallocating the original while borrowed copies exist. In Cyclone, a pointer is borrowed using an explicit `alias` declaration, similar to Walker and Watkins' `let-region` [14], and the LIFO region machinery prevents the borrowed pointer from escaping.

```
fft_state *\U fft_alloc(int n, int inv);
void fft(fft_state *‘r st,...);
void fft_free(fft_state *\U st) consume(1);
void do_fft(int numffts, int inv) {
  fft_state *‘U x = fft_alloc(nfft,inv);
  for (i=0;i<numffts;++i) {
    let alias<‘s> fft_state *‘s a = x;
    fft(a,...);
  }
  fft_free(x);
}
```

Fig. 3. Pointer borrowing example

Consider the example in Figure 3 (adapted from the *KissFFT* benchmark). The `do_fft` function allocates a unique pointer `x` to hold the state of the transform, performs the specified number of FFT's, and then frees the state. The declaration

```
let alias<‘s> fft_state *‘s a = x; ...
```

introduces a fresh region name, `‘s`, and an alias `a` for `x` that appears to be a pointer into region `‘s`. Note that the aliasability `\U` does not appear on the type of `a`. This indicates that within the scope of the `alias` declaration (which is the entire loop body), we may freely copy `a` and pass it to functions that expect aliasable parameters. However, because `‘a` is a fresh name, `a` cannot *escape*; i.e., it cannot be

[6] An allocator supporting such deallocations would let us remove this restriction, though the fact that C allows pointers just beyond allocated objects may complicate matters.

assigned to any variables defined outside the scope of the `alias`, such as a global variable, since those variables cannot be typed as having the region `‘s`. While `a` is live, the original unique pointer `x` is temporarily consumed. This prevents the object to which it refers from being deallocated. At the end of the block, `a` and its copies will be unusable, since the region `‘s` will not be in scope. Thus, no usable aliases can survive the exit from the block, and we can safely restore the type-state of `x` to be an unconsumed, unique pointer, permitting it to be freed with `fft_free`. In short, regions provide a convenient way to temporarily name unique pointers and track aliasing for a limited scope.

We provide a limited but extremely convenient form of `alias`-inference around function calls to simplify programming and cut down on annotations. In particular, whenever a function expecting an aliasable pointer (such as `fft`) is called with a unique pointer as an argument, the compiler will attempt to wrap an `alias` declaration around the call, thereby allowing the argument to be freely duplicated within the callee. As a result, we can rewrite `do_fft` as:

```
void do_fft(int numffts, int inv) {
  fft_state *\U x = fft_alloc(nfft,inv);
  for (i=0;i<numffts;++i) {
    fft(x,...);
  }
  fft_free(x);
}
```

and the compiler will insert the appropriate `alias` declaration for `x`.

3.3 Reference Counting

Even with borrowing, unique pointers can be used only to build tree-like data structures with no internal sharing or cycles. While GC or LIFO arenas may be reasonable options for such data structures, another alternative often employed in systems applications is reference counting. For example, reference counting is used in COM and the Linux kernel, and is a well-known idiom for C++ and Objective C programs.

We found we could elegantly support safe reference counting by building on the discipline of unique pointers. This has two advantages: First, we introduce almost no new language features, rather only some simple run-time support. Second, the hard work that went into ensuring that unique pointers coexisted with conventional regions is automatically enjoyed for reference-counted objects.

An additional aliasability, `\RC` for *reference-counted*, is used to qualify pointers to reference-counted objects; when allocated, these objects are prepended with a

hidden reference-count field. As with unique pointers, the flow analysis prevents the user from making implicit aliases. Instead, a pointer of type `T *\RC` must be copied *explicitly* by using the (built-in) `alias_refptr` function, which increments the reference count and returns a new alias, without consuming the original pointer. This function has the type

```
‘a *\RC‘r alias_refptr(‘a *\RC‘r);
```

In essence, the argument and the returned value both serve as explicit capabilities for the same object. A reference-counted pointer is destroyed by the `drop_refptr` function. This consumes the given pointer and decrements the object's reference count at run-time; if the count becomes zero, the memory is freed.

```
void drop_refptr(‘a *\RC) consume(1);
void cmd_pasv(struct conn *\RC‘H c) {
  struct ftran *\RC f;
  int sock = socket(...);
  f = alloc_new_ftran(sock,alias_refptr(c));
  c->transfer = alias_refptr(f);
  listen(f->sock, 1);
  f->state = 1;
  drop_refptr(f);
}
```

Fig. 4. Reference counting example

Consider the example in Figure 4, adapted from the *Betaftpd* benchmark. In Betaftpd, `conn` structures and `ftran` structures mutually refer to one another, so we chose to manage them with reference counting. Therefore, we must explicitly alias `c` when allocating `f` to point to it. Likewise, we alias `f` explicitly to store a pointer to it in `c`. Once the `sock` connection is established, we no longer need the local copy of `f`, and so we drop it explicitly, leaving the only legal pointer to it via the one stored in `c`.

Thus, treating reference-counted pointers as if they were unique forces programmers to manipulate reference counts explicitly. While this is less convenient than automatic reference counting, it requires almost no additional compiler support. Furthermore, the constraints on unique pointers ensure that an object is never prematurely deallocated, and the flow analysis warns when a pointer is potentially "lost." Finally, we can use the `alias` construct to borrow a reference-counted pointer to achieve a form of explicit, deferred reference counting [15]. Thus, the programmer has complete control over where reference counts are manipulated.

3.4 Reaps

So far we have discussed only unique and reference-counted pointers to heap-allocated objects, but Cyclone supports tracked pointers into other regions, including LIFO arenas. This lets us deallocate some objects in an arena individually, and deallocate the rest of the objects en-masse, with the entire arena. Berger et al. [12] have shown that this strategy can improve space/time performance over traditional arenas, and introduced *reaps* to support it. Cyclone supports a safe form of reaps.

A Cyclone pointer type written out in full has the form

```
T*@aqual(‘q) @region(‘r),
```

where `T` is the pointed-to type, `‘q` is the pointer's aliasability (e.g., `\U`, `\RC`, `\A`), and `‘r` is its region (e.g., `‘H`, or a LIFO arena name). Objects that are referred to by a pointer with `\U` aliasability can be manually deallocated at any time within the lifetime of the region `‘r`, and all remaining objects in region `‘r` are deallocated at its end. Reference counted objects (those objects referred to by pointers with `\RC` aliasability) may also be deallocated prior to the destruction of the entire region as a result of decrementing the reference count. Thus, the lifetime of the region is an upper bound on the lifetime of the object.

Cyclone's general allocation routine is

```
rqmalloc(r,q,sz),
```

where `r` is a region handle, `q` is an *aliasability handle*, and `sz` is the amount of space to allocate (the type of `rqmalloc` is explained in detail in Section 3.6). An aliasability handle has type `aqual_t<‘q>`, where `‘q` is the aliasability of the pointer to be returned. Cyclone provides three aliasibility handle constants:

```
unique_qual : aqual_t<\U>
refcnt_qual : aqual_t<\RC>
 alias_qual : aqual_t<\A>
```

Thus `rqmalloc(h,refcnt_qual,sizeof(int))` allocates a reference-counted integer into the region with handle

`h`. The other allocation routines are defined in terms of `rqmalloc`: `malloc(e)` is shorthand for

```
rqmalloc(heap_region,alias_qual,e)
```

and `rmalloc(h,e)` is shorthand for `rqmalloc(h,alias_qual,e)`.

The functions `rfree(r,e)` and `rdrop_refptr(r,e)` deallocate objects. If `r` has

type region_t<‘r>, then for rfree we require e to have type T *\U‘r, and for rdrop_refptr we require e to have type T *\RC‘r. The free and drop_refptr functions are just shorthands.

We present a simplified example here, taken from the *MediaNet* benchmark, that illustrates one use of reaps.

```
void connectSend(region_t<‘r> r, cmpt_t<‘r> c,
                 conn_t<‘H> cn : single(‘r)){
  struct InportFn *f =
    rqmalloc(r,unique_qual,sizeof(struct InportFn));
  f->fun = writeData;
  f->env = cn;
  f :=: c->outports[0];
  rfree(r,f);
}
```

(The single(‘r) annotation can be ignored for now; it will be revisited in Section 3.8.) Here, c is a component used to perform a stream transformation, and r is the region in which c and its auxiliary data is allocated. The component's outports array is a list of closures that are invoked with the result of the transformation; each closure implemented as a struct InportFn. The connectSend function is invoked for a component whose output closure has been made invalid by a failed network connection. Therefore, it is replaced by a new closure f, whose function writeData will operate on the new, valid connection cn. The old closure is swapped out and freed using rfree. Reaps are a useful paradigm for this example: in the general case, all component data will be freed at once, but repeated network connections will not cause the arena to grow too large.

We modified Cyclone's arena implementation to use (a modified version of) the bget [7] allocator to implement rfree. When an arena is created, the system allocator allocates a chunk of memory from which bget allocates and frees individual objects. Bget consumes two header words per object to record the size of the allocated object, and a pointer to the previous free block. Allocations are implemented by pointer bumping until a deallocation occurs. When objects are deallocated, adjacent freed chunks are aggregated and chained into a free list which is used to serve further allocation requests. Bget acquires more memory from the system if no chunk in the free list is sufficiently large, and frees all the chunks when the arena is deallocated. Note that this allocation strategy is required only for reaps; standard arenas still use a simple pointer-bumping allocation scheme. To distinguish between the two, reaps are allocated using the construct reap<‘r> h as opposed to region <‘r> h. This distinction is also important for ensuring that rfree is sound in the presence of subtyping; we cover this issue in depth in Section 3.8.

[7] http://www.fourmilab.ch/bget/

3.5 Potential Memory Leaks

In general, our analysis does not prevent a programmer from forgetting to free a unique or reference-counted pointer (though it will never allow access through a pointer that has been freed). This is for flexibility and usability. For example, at join points in the control-flow graph, our analysis conservatively considers a value consumed if there is an incoming path on which it is consumed. In particular, if `p` is not consumed and we write:

```
if (rand()) free(p);
```

then the analysis treats `p` as consumed after the statement. In this situation, `p` will leak in the case that true-branch is not taken. As other examples, when we store a unique pointer into a shared object, we may overwrite another live unique pointer. Or when freeing an object referred to by a unique pointer, we may have forgotten to free pointers it contains. Finally, we permit casting a unique pointer to an aliasable one (which obviously cannot be freed).

We considered signalling errors in these cases, as in Vault [16], but found that using exception handlers (to which there are many control-flow paths) and storing unique pointers within shared containers generated too many false alarms. Fortunately, the region into which a unique pointer points serves as a safety-net: if it is the heap, then the leaked object will be GCed, or if it is an arena, then the object will be collected when the arena is freed.

3.6 Polymorphism

In general, the interaction of unique pointers with subtyping and region polymorphism requires some care, as the following function illustrates:

```
`a copy(`a x) { return x; }
```

The syntax `` `a `` denotes a Cyclone *type variable* which different callers can instantiate with different (pointer) types [17], [18]. The `copy` function simply returns its pointer argument as its result. Consider a call to `copy` with a unique pointer:

```
int *\U y, *\U z = malloc(sizeof(int));
y = copy(z);
free(z);
*y = 1; // ERROR!
```

When calling `copy`, `z` is not consumed, but a copy of it is returned and stored into `y`. Thus the caller can free `z` and then erroneously use `y`. To prevent this situation,

we need to distinguish between polymorphic values that can be copied (i.e., in the body of the `copy` function) and those that cannot.

To this end, we distinguish between the aliasability of polymorphic values by employing a form of bounded polymorphism [19] over the aliasabilities of a type. An upper bound on the aliasability of all types that may instantiate a type variable is specified following the type variable. For instance, `'a\A` represents all *aliasable* boxed types (boxed types are pointer types and `int`; unboxed types include all types that are not word-sized, such as a `struct` type) such as `int*` or `int*\U*\A` while `'a\U` represents all *unique* boxed types such as `int*\U`, or `int*\A*\U`. Notice that the bound refers only to the *outer-most* aliasability on a type variable, since only that aliasability should affect how a value is copied.

In our above example, the `copy` function would be typed as:

```
'a\A copy('a\A x);
```

This indicates that `'a` can only be instantiated with aliasable pointers, and so would forbid the call `copy(a)` above, as `a` is unique. (Type variables `'a` have an aliasable bound `\A` by default and need not be annotated explicitly as we have done here.) As another example, consider the type of `rqmalloc`, introduced earlier:

```
'a*@aqual('q) @region('r) rqmalloc(region_t<'r> r,
                                   aqual_t<'q> q,
                                   sizeof_t<'a> s);
```

Thus, `rqmalloc` allocates an object in the region `'r`, returning a pointer of aliasability `'q`.

3.7 Dynamic Arenas

Just as regions make unique pointers more flexible (thanks to the `alias` construct), we found we can use unique pointers to provide a more flexible form of arenas that avoids the LIFO lifetime restriction. The basic idea is to use a unique pointer as a capability or "key" for the arena. The operation `new_ukey()` creates a fresh arena `'r` and returns a unique key for the arena. This key is represented as a unique pointer and has type `uregion_key_t<'r>`. Accessing the arena requires possession of the key, as does deallocating the arena, which is performed by calling `free_ukey()`. Since the key is consumed when the arena is destroyed, and there are no aliases to the key, the arena can no longer be accessed.

Rather than requiring the key be presented on each allocation or pointer-dereference into the arena, we provide a lexically-scoped `open` construct that temporarily consumes the key and allows the arena to be freely accessed within the scope of the

open. The key is then given back upon exit from the scope.

```
trie_t<‘r> trie_lookup(region_t<‘r> r, trie_t<‘r> t,
                       char *‘H buff) {
  switch (t->children) ... // dereferences t
}
int ins_typedef(uregion_key_t<‘r> k,
                trie_t<‘r> t, char *‘H s ; {}) {
  { region h = open(k);  // may access ‘r, not k
    trie_t<‘r> t_node = trie_lookup(h,t,s);
    ...
  } // k unconsumed, ‘r inaccessible
  return 0;
}
```

Fig. 5. Dynamic Arena example

Consider the example in Figure 5, adapted from the Cyclone compiler's lexer: The function `ins_typedef` takes a unique key to some arena `‘r`, along with a pointer `t` to a `trie_t` stored in `‘r`. The annotation "`; {}`" on the function's prototype is an empty "effect." The fact that it is empty effectively denotes that `‘r` need not be live when the function is called (see our earlier paper [4] or the Cyclone manual for more detail on effects). Within the body of `ins_typedef`, `t` may not be dereferenced until it can be shown that the region of `t` is live. (By default, when no effect is indicated, all regions mentioned in a prototype are assumed to be live; the heap region `‘H` is always live.) Introducing `‘r` into the set of live regions is accomplished by the following mechanism for opening an arena. Within the function, the arena is opened via `region h = open(k)`, which adds `‘r` to the set of accessible regions. Thus, `t` can be dereferenced, and the call to `trie_lookup` is safe. The handle `h` permits the user to perform additional allocation into the arena if desired. To prevent the arena from being freed while it is in use, the key `k` is temporarily consumed until the scope concludes, at which time it can be safely destroyed.

Clearly, `open` and `alias` are related. Both provide a way to temporarily "pin" something and give it a name for a particular scope. In the case of `alias`, a single object is being pinned, whereas in the case of `open`, an arena is being pinned. Pinning prevents the object(s) from being deallocated throughout the scope, and the region name is used to prevent pointers to the object(s) from escaping the scope. Thus, while lexically-scoped, LIFO arenas can be limiting, lexically-scoped region names have proven invaluable for making unique pointers and dynamic arenas work well in practice.

As a simple generalization, we support reference-counted arenas, which use reference-counted pointers as keys instead of unique pointers. It is the *key* that is reference-counted, so accessing objects in the arena just requires using the `open` construct with the reference-counted key, as above. When the last reference to a key is dropped, the key and the associated region are deallocated.

3.8 Subtyping

To provide greater flexibility and to promote reuse of code, we support two forms of subtyping over the qualifiers of a pointer: first, subtyping over the alias qualifiers of a pointer; and second, subtyping over the region qualifiers of a pointer.

To support alias qualifier subtyping, we include a fourth aliasability qualifier called `top` which is super-type of the other three qualifiers. None of the other qualifiers are subtypes of each other. The type of a pointer to τ of top-aliasability is denoted τ`*\T`. Such pointers must obey the aliasing restrictions of unique aliasbility pointers, but their referents may not be deallocated.

Subtyping for region qualifiers is defined in terms of the lifetimes of the regions. We say that a region `‘r` *outlives* a region `‘s` if the lifetime of `‘r` is strictly greater than the lifetime of `‘s`. In such a case, it is permissible to use a pointer of type `T *‘r` where a `T *‘s` is expected. The outlives relation induces a partial order on region names. The heap is not oulived by any other region, and, by default, outlives all other regions. LIFO arenas outlive each other according to the LIFO ordering. Dynamic arenas can only outlive other regions when they are open; they may be outlived by only the heap.

A consequence of the outlives relation is that a pointer declared to be of type `T *‘r` may be aliased both by pointers declared to be of type `T *‘a`, where `‘a` outlives `‘r`, as well as by pointers declared to be of type `T *‘b`, where `‘b` is outlived by `‘r`. This relationship is illustrated by the program in Figure 6, where, at some point during the execution of the function, `a`, `b`, and `c` may all be aliases of each other.

```
int region_subtype(uregion_key_t k, int *‘H c) {
   region<‘dyn> dyn = open(k);
   int *‘dyn a;
   { region<‘r> r;
     int *‘r b;
     if(*c) a = c; //‘H outlives ‘dyn
     else   a = rmalloc(dyn, 0);
     b = a; //‘dyn outlives ‘r
   }
}
```

Fig. 6. An illustration of region subtyping

The outlives relation poses a problem when it is applied to reaps. The implementation of `rfree` (introduced in Section 3.4) requires both a region handle and the pointer to the object to be deallocated to be passed as arguments. We further require that the object to be deallocated reside in the region referred to by the handle. The outlives relation allows this constraint to be violated, as in the example below.

```
region<‘r> r;
int *\U ‘H p = rmalloc(heap_region, unique_qual, 0);
rfree(r, p);  // BAD!
```

Since the heap outlives all other regions, the region subtyping rules permit `p` to be used in place of a `int *\U‘r` pointer. Clearly, the object to be deallocated in the call to `rfree` does not reside in the region `‘r`.

There are a number of solutions to this problem. One option is a dynamic check to ensure that pointer passed to `rfree` actually points to a location within the specified region. To avoid the run time overhead, we opted to refine the type system so as to *statically* guarantee that the pointer passed to `rfree` points within the appropriate region. This is achieved by limiting the use of the region subtyping for reaps.

The type of `rfree` is:

```
void rfree(region_t<‘r>, ‘a *\U ‘r : single(‘r)) consume(2);
```

This type states that `rfree` expects a region handle and a pointer of unique aliasability as arguments. The construct `single(‘r)` specifies that the region `‘r` may *not be outlived* by any other region. Finally, the attribute `consume(2)` is a post-condition on the function which states that the pointer passed in the second argument is no longer live, i.e., the referent has been deallocated.

The heap region is clearly satisfies this constraint: `single(‘H)` is always true since the heap is not outlived by any other region. For the other cases, recall that we know which regions may have individually-deallocatable objects: they were declared with the form `reap<‘r> r`.[8] This declaration introduces an assertion of the form `single(‘r)` into the type-checking environment, which prevents the use of the outlived-by relation for any pointer to `‘r`. It also allows the `single` constraint in the type of `rfree` to be proved. This is why the prototype for the *MediaNet* example from Section 3.4 required the `single(‘r)` constraint in its prototype: this is necessary to be able to correctly call `rfree`.

3.9 Summary

The addition of unique pointers to our region framework gives us a number of memory-management options, which are summarized in Tables 1 and 2. Clearly, programmers have a variety of options, particularly in choosing how to deallocate objects. However, we emphasize that the easy default of using the garbage collector

[8] We impose a similar requirement on the creation of dynamic arenas that support the deletion of individual objects–the key for such an arena must be created using the function `new_reap_ukey`.

<table>
<tr><th rowspan="2">Region Variety</th><th rowspan="2">Allocation</th><th colspan="2">Deallocation</th><th rowspan="2">GC</th></tr>
<tr><th>Per Object</th><th>Whole Region</th></tr>
<tr><td>Stack</td><td>static</td><td>no</td><td rowspan="2">on exit of</td><td rowspan="4">no</td></tr>
<tr><td>LIFO Arena</td><td rowspan="3">dynamic</td><td rowspan="2">for unique</td></tr>
<tr><td>Dynamic Arena</td><td rowspan="2">and refcnt</td><td>manual</td></tr>
<tr><td>Heap</td><td>pointers</td><td>never</td><td>yes</td></tr>
</table>

Table 1
Summary of Memory-Management Options in Cyclone

is always available. As the alias restrictions indicate, a particular choice essentially trades ease of use for better control over object lifetime. Each choice encodes a sound static discipline: Only reference counting and swaps have run-time cost.

4 Applications

Table 3 describes the benchmark programs with which we experimented. For programs we ported from C (Boa, Betaftpd, Epic, KissFFT, Cfrac, 8139too, i810_audio, and pwc) it shows the non-comment lines of code of the original program, and then the changes due to porting to Cyclone without the use of any manual memory management mechanisms, and then the additional changes needed to use manual mechanisms. For the first five programs, in the absence of manual memory management we used heap allocation, managed by GC. The next three C programs are Linux device drivers—for these, we could not link a garbage collector into the kernel, so the non-manual versions heap-allocate but never free anything; we provide this figure only to illustrate the additional (programming) effort of using manual techniques.

The other programs (MediaNet, CycWeb, and CycScheme) were written directly in Cyclone. The final column indicates which manual mechanisms we used. For all programs other than MediaNet, we could easily eliminate the need for GC. This section presents our experience using Cyclone to port or write these programs, considering first the standalone C programs, then the device drivers, and finally the Cyclone-only programs. Performance experiments for these programs are presented in the next section.

4.1 Standalone C programs

The process of porting from C to Cyclone is made easiest by placing all dynamically-allocated data in the heap region and letting the GC take care of recycling the data.

Function/Syntax	Description
`rqmalloc`	Generic allocation (§3.4, §3.6)
`rmalloc`, `malloc`	(`\A`) Aliasable pointer allocation (§2.1, §3.4)
`free`, `rfree`	(`\U`) Unique pointer deallocation (§3.1, §3.4)
`alias_refptr`, `drop_refptr`, `rdrop_refptr`	(`\RC`) Increment/decrement reference counts (§3.3)
`new_ukey`,`new_rckey`	Allocate dynamic arenas (§3.7)
`new_reap_ukey`, `new_reap_rckey`	Allocate dynamic reaps (§3.4,§3.8)
`free_ukey`,`free_rckey`	Free dynamic arenas or reaps (§3.7)
`e1 :=: e2`	Swap contents of `e1` and `e2` (§3.1)
`let alias<‘r> T x = e; s`	Temporarily alias tracked pointer `e` as `x` in `s` (§3.2)
`region<‘r> h; s`	Make LIFO arena with handle `h`, live in `s` (§2.1)
`reap<‘r> h; s`	Make LIFO reap with handle `h`, live in `s` (§3.4,§3.8)
`region<‘r> h = open(d); s`	Make dynamic arena with key `d` accessible via handle `h` within `s` (§3.7)

Table 2
Summary of Memory Management Functions and Syntax

Most of the changes involve differences between C and Cyclone that are not related to memory management, such as introducing *fat* pointer annotations. (Fat pointers carry run-time array-bounds information with them to support dynamic bounds checks when pointer arithmetic cannot be statically verified.) To take advantage of the new manual facilities, we roughly performed two actions. First, we distinguished data with scoped lifetime from other data. Second, for those data structures with a non-scoped lifetime, we identified their aliasing behavior to determine which mechanism to use.

Objects with Scoped Lifetime When data structures have scoped lifetimes, we can either allocate them in a LIFO arena, or we can allocate them in the heap as unique pointers, and use the `alias` construct to allow temporary aliasing until they can be freed. For example, in both Epic and KissFFT, we merely had to change a declaration from something like

```
T* q_pyr = malloc(...);
```

to instead be

```
T*\U q_pyr = malloc(...);
```

Program	Description	Non-comment Lines of Code			Manual
		C	Cyc	Cyc (+manual)	mechs
Boa	web server	5217	± 286 (5%)	± 98 (1%)	U
Cfrac	integer factorization	3143	± 183 (5%)	± 784 (25%)	UD
Betaftpd	ftp server	1164	± 219 (18%)	± 238 (22%)	UR
Epic	image compression	2123	± 218 (10%)	± 117 (5%)	UL
KissFFT	fast Fourier transform	453	± 74 (16%)	± 25 (5%)	U
8139too	RealTek Ethernet driver	1972	± 971 (49%)	± 312 (14%)	UD
i810_audio	Intel sound driver	2598	± 1500 (57%)	± 318 (10%)	RD
pwc	Philips webcam driver	3755	± 1373 (36%)	± 1024 (26%)	RD
MediaNet	streaming overlay network		8715	± 320 (4%)	URLD
CycWeb	web server			667	U
CycScheme	scheme interpreter			2523	ULD

U = unique pointers R = ref-counted pointers L = LIFO regions D = dynamic arenas

Table 3
Benchmark Programs

All `alias` statements were inferred automatically when calling functions that wished to alias the array (see Figure 3). For Epic, we also used a LIFO arena to store a Huffman compression tree that was used during the first phase of the compression. This required changing the prototypes of the creator functions to pass in the appropriate arena handle, in addition to adding various region annotations (see Figure 1).

Objects with Dynamic Lifetime If we wish to manage a data structure manually using unique pointers, it cannot require aliases. For example, Boa stores the state of each request in a `request` structure, illustrated in Figure 7. Because these form a doubly-linked list, we cannot use unique pointers. Even if we were to remove the `next` and `prev` fields and store `request` objects in a separate (non-unique) list, we could not uniquely allocate requests because they contain internal aliases. For example, the `header` field identifies the HTTP header in an internal buffer.

In the case of Boa, `request` objects are managed by a custom allocator. Throughout its lifetime, a request moves between a blocked queue and a ready queue, and when complete, the request moves onto a free list to be reused. Therefore, we can continue to use this allocator and simply heap-allocate the requests since they will never be freed by the system allocator. Some internal elements of the request, such as the pathname (shown with open-headed arrows in the figure) were not aliased internally, so they could be uniquely allocated and freed when the request was

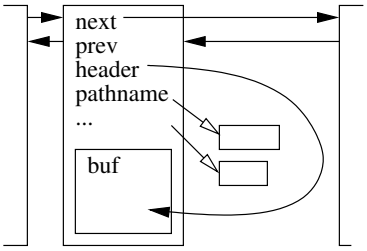

Fig. 7. Request data structure in Boa

complete.

Betaftpd also used doubly-linked lists, one for open sessions and one for transfers in progress. Furthermore, there are cross-links between session and transfer objects. Thus, reference-counted pointers seemed like the best safe option that avoided garbage collection. As Table 3 shows, this required changing 22% of the code. The reason is that all reference counts are managed manually, so we had to insert many calls to `alias_refptr` and `drop_refptr` along with the addition of annotations. While largely straightforward, we were forced to spend some time tracking down memory leaks that arose from failing to decrement a count. The warnings issued by the compiler were of little help, since there were too many false positives. However, we remark that the original program had a space leak along a failure path that we were able to find and eliminate.

Cfrac also required significant changes. The original C program made use of application-level reference counting, but we found it both more natural and more efficient to use a combination of dynamic arenas and unique pointers. The program is structured around a few long running loops that dynamically allocate data with each iteration. However, only a small amount of the allocated state is carried over to the next iteration. The Cyclone version allocates within a dynamic region and, prior to the next iteration of the loop, copies the relevant state to a new dynamic region before freeing the old one. A large portion of the changes were due to the need to pass the region handle to functions so that no allocation is performed in the heap.

Note that because the changes to legacy code did not change the underlying data structures or logic of the application (though we did sometimes change how data structure memory was managed), we generally avoided introducing application-level bugs while porting. Straightforward testing worked well, using the original C program as the ground truth, though the inherent "danger" of changing existing code cannot be completely eliminated.

4.2 Linux Device Drivers

We ported to Cyclone three device drivers, for Linux version 2.4.21. The basic process has two parts. First, we must port the code within the driver to be safe,

and second, we must give Cyclone types to functions defined in the kernel that are called by the driver. This latter task was made simpler by a custom tool written to ensure that the Cyclone type is representation consistent with the C type it is meant to represent. In some cases, we could not call kernel functions directly while remaining type safe, but rather needed to write wrapper functions that performed extra checks.

Much of the effort involved in porting the code within a device driver to Cyclone is due to ensuring that array accesses are within bounds, particularly accesses to I/O buffers. While in a standalone application we could quickly and easily use fat pointers for this purpose, this was rarely an option in the kernel. This is because we were constrained to preserve the layout of many kernel data structures, using one word for a pointer, where fat pointers would require three words. As a result, we had to use more stylized and verbose idioms (in particular, a form of dependent products [20]) to permit the compiler to prove that array accesses were always in bounds.

Choosing a manual memory management strategy for device drivers was also made more challenging by the kernel environment. Compared to porting an application program, there were three challenges. First, it is generally infeasible, and arguably undesirable, to use garbage collection in the kernel. Second, it is not possible to redefine the representation of data structures passed to and from the kernel. Finally, device drivers use pointer arithmetic extensively to access and modify buffers. Pointers to arbitrary offsets within these buffers are also often maintained. This sometimes makes using tracked pointers difficult or impossible.

In the end, we found that dynamic arenas containing aliasable pointers, coupled with occasional use of unique and reference-counted pointers, often worked the best. Most drivers allocate a data structure that is opaque to the kernel, though it may be referenced by kernel data structures. We typically changed this structure to include a dynamic arena handle, using that to store any allocated memory. When the device is unloaded, the dynamic arena is freed. We adapted the Cyclone runtime to use the kernel allocation functions (kmalloc and vmalloc) for obtaining and freeing region pages; this and other runtime support (e.g., to support exceptions and bounds checking) was provided by an additional module implemented in C.

In addition to using kernel memory, a device driver may also manipulate I/O memory that resides on the device. To ensure that this memory is managed correctly, we wrote wrappers for the routines that allocate and release I/O buffers, reusing some of the infrastructure of dynamic arenas. In particular, the allocation wrapper returns the allocated buffer together with a *capability*, in the form of a dynamic arena's key, that establishes the availability of the buffer. As long as the key is live, the I/O memory is live; when the deallocation wrapper frees the memory, it consumes the key as well. Figure 8 shows a code snippet from 8139too that illustrates this idiom.

```
typedef struct pci_mem { <`r>
  uregion_key_t<`r> *key;
  char *`r buf;
} pci_mem_t;
char *`H pci_alloc_consistent(pci_dev_t, int, dma_addr_t);
pci_mem_t cyc_pci_alloc(pci_dev_t hwdev, int len,
                        dma_addr_t *dma_handle) {
  let NewDynamicRegion<`d> key = new_ukey();
  let iobuf = pci_alloc_consistent(hwdev, len, dma_handle);
  return pci_mem{new key, iobuf};
}
```

Fig. 8. Usage of dynamic arenas to represent I/O memory

The kernel function `pci_alloc_consistent` allocates a buffer from memory that resides on the device. The type of this function asserts that the return value is a pointer into the heap. We also want to be able to support the deletion of the buffer, say when the driver is unloaded. The driver, however, maintains many aliases to arbitrary offsets within the allocated buffer. Hence it is not possible to treat the returned value as a unique pointer to the allocated data.

Therefore we provide `cyc_pci_alloc`, a Cyclone wrapper for the allocation of I/O memory. The wrapper models each buffer allocated in I/O memory as if it resided in a distinct dynamic arena. First, it invokes `new_ukey` to generate a new dynamic arena for the memory. No actual kernel memory is allocated by this call, since Cyclone's arena implementation delays the allocation of the memory pages of an arena until the first allocation into that arena. Next, the buffer is allocated on the device using a standard invocation of the kernel function `pci_alloc_consistent`, which is initially treated as if it points into the heap. Then we coerce the pointer to be as if it pointed into dynamic arena `` `d ``, which is legal since the heap outlives all other regions. The pointer to the buffer and the region key are then packaged as a `pci_mem` object, in which the name `` `r `` is existentially quanitified. Code that wishes to use the allocated buffer must also present the key of the dynamic arena, as in the example of Figure 5. The corresponding deallocation function, `pci_free_consistent`, is also wrapped so that the key is consumed.

4.3 Cyclone Applications

In addition to porting C programs, we have written three Cyclone programs from scratch that use our manual mechanisms.

CycWeb CycWeb is a simple, space-conscious web server that supports concurrent connections using non-blocking I/O and an event library in the style of libasync

[21] and libevent [22]. The event library lets users register *callbacks* for I/O events. A callback consists of a function pointer and an explicit environment that is passed to the function when it is called. The event library uses polymorphism to allow callbacks and their environments to be allocated in arbitrary regions. For the library, this generality is not overly burdensome: of 260 lines of code, we employ our swap operator only 10 times across 10 functions, and never use the `alias` primitive explicitly. The web server itself (667 lines) has 16 swaps and 5 explicit `aliass`.

For the rest of the application, we also chose to use unique pointers. When a client requests a file, the server allocates a small buffer for reading the file and sending it to the client in chunks (default size is 1 KB). Callbacks are manually freed by the event loop when the callback is invoked (they must be re-registered if an entire transaction is not completed); each callback is responsible for freeing its own environment, if necessary. As we see in the next section, this design allows the server to be reasonably fast while consuming very little space.

MediaNet MediaNet [23] is an overlay network with servers that forward packets according to a reconfigurable DAG of *operations*, where each operation works on the data as it passes through. For better performance, we eschew copying packets between operations unless correctness requires it. However, the dynamic nature of configurations means that both packet lifetimes and whether packets are shared cannot be known statically.

```
struct StreamBuff { <i::I>
  ... // three omitted header fields
  tag_t<i> numbufs;
  struct DataBuff<\RC> bufs[numbufs];
};
struct DataBuff<'q> {
  unsigned int ofs;
  byte ? 'q buf;
};
```

Fig. 9. MediaNet packet data structures

We use a data structure called a *streambuff* for each packet, similar to a Linux `skbuff`, shown in Figure 9. The packet data is stored in the array `bufs`. Note that `bufs` is not a pointer to an array, but is flattened directly within `StreamBuff`. Thus `StreamBuff` elements will vary in size, depending on the number of buffers in the array. The `numbufs` field holds the length of `bufs`. The notation <i`::I>` introduces an *existential* type variable of integer kind (`I`) [17], and is used by our type system to enforce the correspondence between the `numbufs` field and the length of the `bufs` array in a fashion similar to Xi and Pfenning's Dependent ML [20]. We often used this paradigm in our device driver ports as well.

Databuffs store packet data. The `buf` field points to an array of the actual data.

The ? notation designates a fat pointer to a dynamically-sized buffer. The `ofs` field indicates an offset, in bytes, into the `buf` array. This offset is necessary since pointer arithmetic is disallowed for unique and reference-counted pointers.

Each `StreamBuff` object is allocated in the heap with unique aliasability. When a packet must be shared, a new streambuff is created, whose array points to the same databuffs as the original (after increasing their reference counts). This approach allows for quickly appending and prepending data to a packet, and requires copying packet buffers only when they are both shared and mutated.

An example using streambuffs is shown in Figure 10. Here, three individual streambuffs A, B, and C share some underlying data; unique pointers have open arrowheads, while reference-counted ones are filled in. This situation could have arisen by (1) receiving a packet and storing its contents in A; (2) creating a new buffer B that prepends a sequence number 1234 to the data of A; and (3) stripping off the sequence number for later processing (assuming the sequence number's length is 4 bytes). Thus, C and A are equivalent. When we free a streambuff, we decrement the reference counts on its databuffs, so they will be freed as soon as possible.

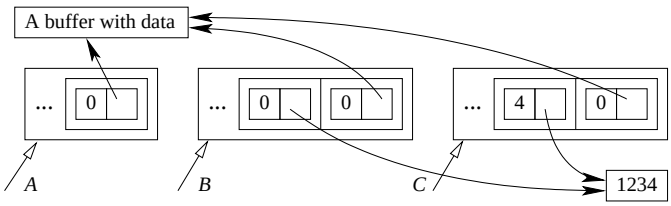

Fig. 10. Pointer graph for three streambuffs

MediaNet's DAG of operations is stored in a dynamic arena. We were able to treat this dynamic arena as a reap in which we allocated objects representing connections. In the event a connection fails the object is deleted from the reap.

An earlier version of MediaNet stored all packet data in the heap, using essentially the same structures. One important difference was that databuffs contained an explicit `refcnt` field managed by the application to implement copy-on-write semantics. Unfortunately, this approach yielded a number of hard-to-find bugs due to reference count mismanagement. Our language support for reference counting eliminated the possibility of these bugs, and further let us free the data immediately after its last use. As shown in Table 3, moving to explicit unique pointers and dynamic regions was not difficult; only 4% of the code had to be changed. The majority of these changes were in two utility files. Out of nearly 9000 non-comment lines, we used swap 74 times and `alias` 67 times, of which 67% were automatically inferred.

CycScheme Using a combination of our dynamic arenas and unique pointers, Fluet and Wang [24] have implemented a Scheme interpreter and runtime system in Cyclone. The runtime system includes a copying garbage collector in the style

of Wang and Appel [25], written entirely in Cyclone. All data from the interpreted program are allocated in a dynamic arena, and when the collector is invoked, the live data are copied from one arena to another, and the old arena is then deallocated. Since both arenas must be live during collection but their lifetimes are not nested, LIFO arenas would not be sufficient. Further details on CycScheme's performance and implementation can be found in Fluet and Wang's paper.

5 Performance Experiments

To understand the benefit of our proposed mechanisms, we compared the performance of the GC-only versions of our sample user-level applications to the ones using manual mechanisms. Where our applications were ported from C, we also compare the performance of the Cyclone ports to the original C versions. Our measurements exhibit three trends. First, we found that elapsed time is similar for the GC and manual versions of the programs. Indeed all our benchmark programs, other than MediaNet, have execution time performance virtually the same for the GC and non-GC cases. In the case of MediaNet, judicious use of manual mechanisms significantly reduced the reliance on GC (but did not eliminate it entirely), improving performance. Second, we found that we could significantly reduce memory utilization by using manual mechanisms. In the cases where the benchmark was ported from C, the use of manual techniques caused the memory consumed by the Cyclone program to be close to that of the C program. Finally, we observed that the increase (if any) in elapsed time for the Cyclone version of a program, relative to its C counterpart, is largely due to the overhead of bounds checking.

In this section we carefully discuss the performance of the Boa, CycWeb, and MediaNet servers. We found these to be the most interesting programs from a resource management point of view; measurements for the remaining programs can be found in Section 5.3. We ran our performance experiments on a cluster of dual-processor 1.6 GHz AMD Athlon MP 2000 workstations each with 1 GB of RAM and running Linux kernel version 2.4.20-20.9smp. The cluster is connected via a Myrinet switch.

We used Cyclone version 0.9 which, along with the benchmarks, is publicly available [26]. By default, Cyclone programs use the Boehm-Demers-Weiser (BDW) conservative collector [3], version 6.4, for garbage collection and manual deallocation. BDW uses a mark-sweep algorithm, and is incremental and generational. We used the default initial heap size and heap-growth parameters for these experiments. We disabled padding in the gc allocator and disabled support for Java. All other configuration options were the default. When programs do not need GC, they are compiled with the Lea general-purpose allocator, [9] version 2.7.2. Cyclone com-

[9] `http://gee.cs.oswego.edu/dl/html/malloc.html`

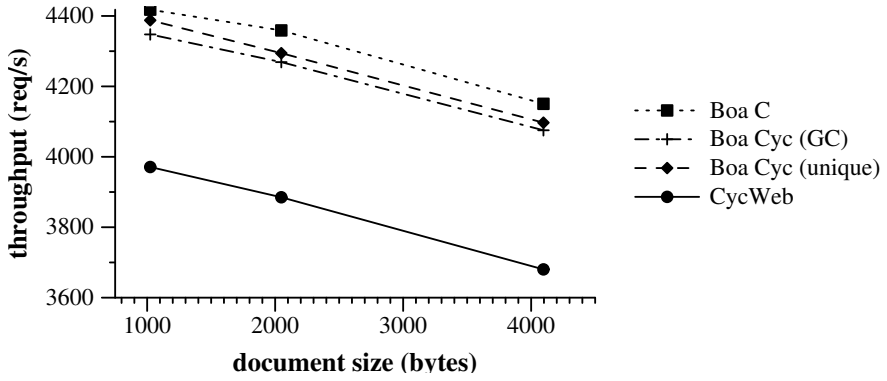

Fig. 11. Throughput for CycWeb and Boa

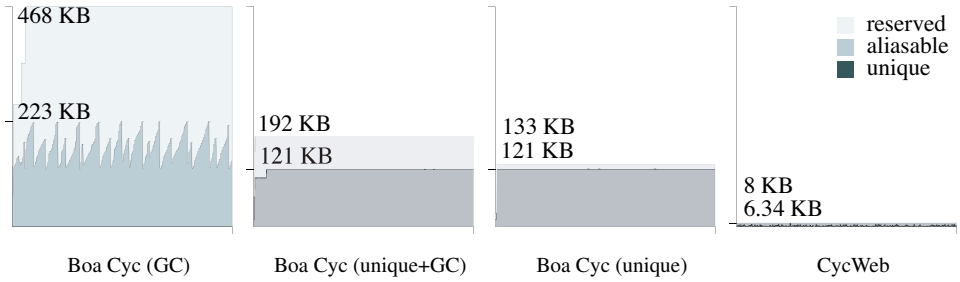

Fig. 12. Memory footprint of Cyc Boa versions

piles to C. All C code was compiled with `gcc` version 3.2.2, at optimization level `-O3`.

5.1 Boa and CycWeb

We measured web server throughput using the SIEGE web benchmarking tool [10] to blast Boa with repeated requests from 6 concurrent users for a single file of varying size for 10 seconds. (We chose 6 users because we observed it maximized server performance.) The throughput results are shown in Figure 11—note the non-zero y-axis. This shows three versions of Boa—C, Cyclone using GC, Cyclone without GC (labeled "unique")—and the single version of CycWeb. We plot the median of 15 trials (the variance was insignificant, and is not shown). For Boa, the Cyclone versions are roughly 2% slower than the C version, with the difference between them negligible (often within the range of error or close to it). Thus, for the performance metric of throughput, removing the GC has little payoff. CycWeb is optimized for memory footprint instead of speed, but is only 10–11% slower than Boa in C.

Avoiding GC has greater benefit when considering memory footprint. Figure 12 depicts three traces of Boa's memory usage while it serves 4 KB pages to 6 concurrent users. The first trace uses GC only, while the others make use of unique

[10] `http://joedog.org/siege/`

pointers. The second (unique+GC) uses BDW as its allocator (thus preventing inadvertent memory leaks), while the third uses the Lea allocator.[11] The x-axis is elapsed time, while the y-axis plots memory consumed. The graph shows memory used by aliasable and unique pointers into the heap, as well as the total space reserved by the allocator (i.e., acquired from the operating system).

The working set size of all versions is similar, and is dominated by the heap region since the majority of memory is consumed by the heap-allocated request structures. The GC version's footprint fluctuates as request elements are allocated and collected (there are 23 total GCs in this case). To ensure reasonable performance, the collector reserves a fair amount of headroom from the OS: 468 KB. By contrast, the unique versions have far less reserved space, with the Lea allocator having little more than that required by the application. We have done memory traces with other file sizes and levels of concurrent access and found the trends to be similar. Very little data is managed as unique pointers (it is not really visible in the graph)—only about 50 bytes per request. In the GC case, this same data is allocated in the heap, and accumulates until eventually collected.

Turning to CycWeb, which uses only the Lea allocator and no garbage collector, we see that we have succeeded in minimizing memory footprint: the working set size is less than 6.5 KB. This is proportional to the number of concurrent requests—we process at most 6 requests at a time, and allocate a 1 KB buffer to each request.

5.2 MediaNet

All the versions of Boa perform very little allocation per transaction, thanks to the use of a custom allocator. The benefit of the allocator depends in part on the fact that `request` objects are uniformly-sized: allocations merely need to remove the first element from the free list. The same approach would work less well in an application like MediaNet, whose packets vary widely in size (from a tens of bytes to tens of kilobytes). Avoiding excessive internal fragmentation would require managing multiple pools, at which point a general-purpose allocator seems more sensible, which is what we used. However, we found that this choice can lead to significant overhead when using GC.

In a simple experiment, we used the TTCP microbenchmark[12] to measure MediaNet's packet-forwarding throughput and memory use for varying packet sizes. We measured two configurations. *GC+free* is MediaNet built using unique and reference-counted pointers for its packet objects (as described above), while *GC only* stores all packet objects in the garbage-collected heap.

[11] The throughput of both versions is essentially the same, so only one line is shown in Figure 11.

[12] `http://ftp.arl.mil/~mike/ttcp.html`

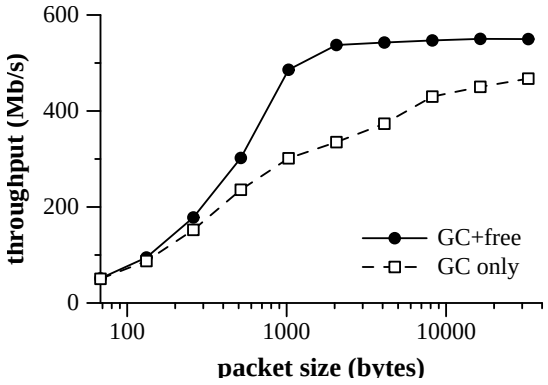

Fig. 13. MediaNet throughput

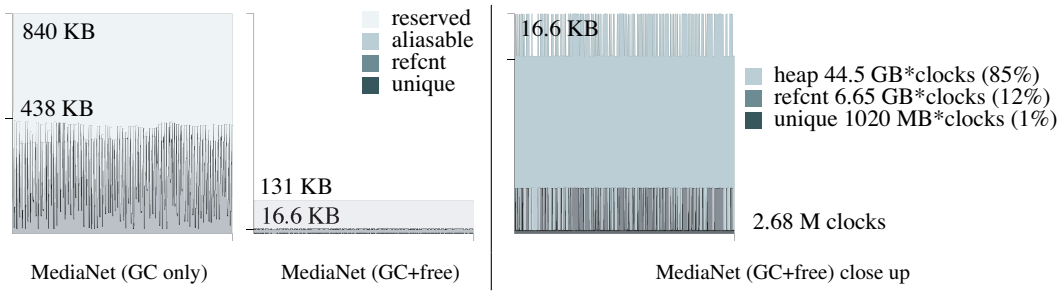

Fig. 14. MediaNet memory utilization

Figure 13 plots the throughput, in megabits per second, as a function of packet size (note the logarithmic scale). Each point is the median of 21 trials in which 5000 packets are transferred; the variance was insignificant. The two configurations perform roughly the same for the smallest packet sizes, but *GC only* quickly falls behind as packets reach 256 bytes. The `GC+free` curve peaks at 2 KB packets, while the *GC only* case never reaches this point; *GC+free* achieves 60% better throughput in the best case. This experiment illustrates the benefit of being able to free a packet immediately. While more sophisticated garbage collectors could well close the gap, the use of manual mechanisms can only be of help. Moreover, even advanced GCs will do less well when packet lifetimes vary due to user processing in the server; our use of reference counting allows packets to be shared and freed immediately when no longer of interest.

Figure 14 illustrates the memory usage of MediaNet when forwarding 50,000 packets, each of size 4KB. This graph has the same format as the graph in Figure 12; only heap-allocated data is shown (the dynamic region in which the configuration is stored never changes and so is uninteresting). The *GC only* configuration stores all data as aliasable (GC'ed) pointers, and so exhibits a sawtooth pattern with each peak roughly coinciding with a garbage collection (there were 551 total on this run). The *GC+free* configuration uses and reserves far less memory: 131 KB as opposed to 840 KB for reserved memory, and 16.6 KB as opposed to 438 KB of peak used memory. There is about 10 KB of initial heap-allocated data that remains throughout the run, and the reference-counted and unique data never consumes more than a single packet's worth of space, since each packet is freed before the next packet

is read in. This can be seen in the close-up at the right of the figure. The topmost band is the aliasable data (the reserved space is not shown), while the feathery band below it is the reference-counted data, with the unique data at the bottom.

5.3 Other Benchmarks

Test	C		Cyclone GC		Cyclone Manual	
	Time	Mem	Time	Mem	Time	Mem
Epic	0.70	12.5M	1.11 (1.61)	22.3M (1.78)	1.11 (1.61)	12.5M (1.0)
KissFFT	1.33	394K	1.40 (1.05)	708K (1.80)	1.41 (1.06)	392K (0.99)
Betaftpd	4.00	6.2K	4.00 (1.0)	192K (30.1)	4.00 (1.0)	8.2K (1.32)
Cfrac	8.75	284K	15.23 (1.74)	1.44M (5.19)	14.53 (1.66)	706K (2.49)
8139too	334	27.7K			333(0.99)	31.8K (1.14)

Table 4
Benchmark performance

The elapsed time measurements of the other benchmarks (reported in seconds) are presented in Table 4. All numbers reported are the median of eleven trials. The variance is not significant. We show the elapsed time relative to the C version in parentheses. To benchmark Epic we used it to encode a 20MB image file; KissFFT came with a benchmarking program and we used it to perform 10,000 transforms; Betaftpd was benchmarked by performing one hundred transfers of a 1MB file using wget; Cfrac was used to factor five 30-digit numbers chosen arbitrarily; 8139too was benchmarked by performing a 3GB transfer using TTCP, where only the sender used the device driver.

In general, the table shows trends consistent the server applications discussed earlier: (1) Cyclone's run time performance is relatively close to that of C; (2) relative to versions that use a GC, Cyclone programs tuned to use manual techniques have significantly lower memory usage, close to that of the C original. In one case (cfrac) we observed an improvement in elapsed time when comparing to the GC case.

Betaftpd is mostly I/O-bound and its timing behavior is therefore insensitive to porting to Cyclone. This is also the case for the 8139too ethernet driver benchmark. We do not report performance benchmarks for the other two drivers except to note that the no noticeable degradation was observed in common usage.

The slowdown of the remaining Cyclone programs relative to C is due, in large part, to dynamic bounds checks for array accesses. For Epic, KissFFT, and cfrac, we also measured the elapsed time of the manual version with bounds checks disabled. KissFFT exhibited identical performance to the C version, and Epic actually got faster by about 10%. Cfrac was still about 40% slower, as compared to 66% with

bounds checks enabled. As described in Section 4.1, Cfrac performs the additional work of copying loop-carried state from one dynamic region to another, and we suspect this is a large part of the remaining overhead.

For the standalone programs, the memory usage reported in Table 4 (column "Mem") is the maximum size of the heap's reserved space during a program run (in bytes). We show the footprint relative to the C version in parentheses. In the best case (Betaftpd) we see a thirty-fold reduction in memory usage, and in one case the reduction amounts to 10 MB of memory (Epic). The worst case is Cfrac, which is more than twice the size of the C version, though still roughly half that of the Cyclone GC version. We suspect the additional overhead is due to the need to copy between arenas with each loop iteration. All the C and manual Cyclone versions of the user-level benchmarks use the Lea allocator. None of the four benchmarks used reaps, so bget was disabled and a pointer bumping allocator was used for region allocations. For the ethernet driver we report the memory usage as the size of the module reported by the `lsmod` command. The difference is due to the greater code size of the Cyclone version.

In summary, this section has demonstrated performance trends that are consistent with other studies comparing GC and manual memory management [27,28]: memory footprint can be reduced, sometimes significantly, and elapsed time can occasionally be improved, particularly when allocation occurs on the critical path. Overall, we believe there is value in using safe manual mechanisms on their own, and as an effective complement to GC. They give programmers more control over the performance of their programs, in many cases without undue programming burden, and without need to compromise safety.

6 Related Work

The ML Kit [29] implements Standard ML with (LIFO) arenas. Type inference is used to allocate data into arenas automatically. Various extensions have relaxed some LIFO restrictions [30,31], but unique pointers have not been considered.

The RC language and compiler [32] provide language support for reference-counted regions in C. However, RC does not prevent dangling pointers to data outside of regions and does not provide the safety guarantees of Cyclone.

Use-counted regions [33] are similar to our dynamic arenas, except there are no alias restrictions on the keys and there is an explicit "freeregion" primitive. Freeing an accessible (opened) region or opening a freed region causes a run-time exception. The remaining problem is managing the memory for the keys. One solution, also investigated in an earlier Cyclone design, is to allocate one region's key in another region, but the latter region typically lives so long that the keys are a space

leak (although they are small). A second solution allows a key to be allocated in the region it represents by dynamically tracking all aliases to the key and destroying them when the region is deallocated. This approach requires more run-time support and cannot allow keys to be abstracted (via polymorphism, casts to `void*`, etc.).

Work on linear types [6], alias types [8,34], linear regions [14,35], and uniqueness types [36] provide important foundations for safe manual memory management on which we have built. Much of this foundational work has been done in the context of core functional languages and does not address the range of issues we have.

Perhaps the most relevant work is the Vault project [16,37] which also uses regions and linearity. Unique pointers allow Vault to track sophisticated type states, including whether memory has been deallocated. To relax the uniqueness invariant, they use novel *adoption* and *focus* operators. Adoption lets programs violate uniqueness by choosing a unique object to own a no-longer-unique object. Deallocating the unique object deallocates both objects. Compared to Cyclone's support for unique pointers in non-unique context, adoption prevents more space leaks, but requires hidden data fields so the run-time system can deallocate data structures implicitly. Focus (which is similar to

Aiken et al's `restrict` [38] allows adopted objects to be temporarily unique. Compared to *swap*, focus does not incur run-time overhead, but the type system to prevent access through an unknown alias requires substantially more user annotations because it must ensure a non-local invariant.

Unique pointers and related restrictions on aliasing have received considerable attention as extensions to object-oriented languages. Clarke and Wrigstad [7] provide an excellent review of related work and propose a notion of "external uniqueness" that integrates unique pointers and ownership types. Prior to this work, none of the analogues to Cyclone's `alias` allowed aliased pointers to be stored anywhere except in method parameters and local variables, severely restricting code reuse. Clarke and Wrigstad use a "fresh owner" to restrict the escape of aliased pointers, much as Cyclone uses a fresh region name with `alias`. Ownership types differ from our region system most notably by restricting which objects can refer to other objects instead of using a static notion of accessible regions at a program point.

Little work on uniqueness in object-oriented languages has targeted manual memory management. A recent exception is Boyapati et al.'s work [39], which uses regions to avoid some run-time errors in Real-Time Java programs [40]. As is common, this work uses "destructive reads" (an atomic swap with `NULL`) and relies on an optimizer to eliminate unnecessary writes of `NULL` on unique paths. Cyclone resorts to swaps only for unique data in nonunique containers, catching more errors at compile time. Few other projects have used swap instead of destructive reads [41,42]. Alias burying [10] eschews destructive reads and proposes using static analysis to prevent using aliases after a unique pointer is consumed, but the

details of integrating an analysis into a language definition are not considered.

Berger et al. introduced the idea of a *reap* [12], which is an arena that supports individual deallocation in addition to the traditional bulk deallocation. Our reaps are safe (avoiding dangling pointer dereferences), and additionally support object deallocation by reference counting.

7 Conclusions

Cyclone supports a rich set of safe memory-management idioms:

- *Stack/LIFO Arenas:* works well for lexically-scoped lifetimes.
- *Dynamic arenas:* works well for aggregated, dynamically allocated data.
- *Uniqueness:* works well for individual objects as long as multiple references aren't needed within data structures.
- *Reference counting:* works well for individual objects that must be shared, but requires explicit reference count management.
- *Reaps:* works well in resource constrained situations and for temporary storage of dynamically allocated data.
- *Garbage collection:* provides simple, general-purpose memory management.

Programmers can use the best idioms for their application. In our experience, all idioms have proven useful for improving some aspect of performance.

This array of idioms is covered by the careful combination of only two linguistic features: regions with lexically-scoped lifetimes and unique pointers. Unique pointers give us the power to reason in a flow-sensitive fashion about the state of objects or arenas and to ensure that safety protocols, such as reference counting, are enforced. Regions work well for stack allocation and give us a way to overcome the burdens of uniqueness for a limited scope.

Nonetheless, there are many open issues that require further research. For instance, a strict, linear interpretation of unique pointers instead of our relaxed affine approach would have helped to avoid the leaks that we encountered and perhaps avoid the need for GC all together. However, we found that the strict interpretation generated too many false type errors in the presence of exceptions and global data.

Another area where further work is needed is in tools to assist the porting process. We generally found that developing new code in Cyclone was easier because we could start with the invariants for a particular memory-management strategy in mind. In contrast, porting legacy code required manually extracting these invariants from the code. Our hope is that we can adapt tools from the alias and shape analysis community to assist programmers in porting applications.

References

[1] M. Hicks, G. Morrisett, D. Grossman, T. Jim, Experience with safe manual memory-management in Cyclone, in: Proc. International Symposium on Memory Management, 2004, pp. 73–84.

[2] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang, Cyclone: A safe dialect of C, in: Proc. USENIX Annual Technical Conference, 2002, pp. 275–288.

[3] H.-J. Boehm, M. Weiser, Garbage collection in an uncooperative environment, Software – Practice and Experience 18 (9) (1988) 807–820.

[4] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in Cyclone, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2002, pp. 282–293.

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

[6] P. Wadler, Linear types can change the world!, in: Programming Concepts and Methods, 1990, IFIP TC 2 Working Conference.

[7] D. Clarke, T. Wrigstad, External uniqueness is unique enough, in: European Conference on Object-Oriented Programming (ECOOP), 2003, pp. 176–200.

[8] F. Smith, D. Walker, G. Morrisett, Alias types, in: Proc. European Symposium on Programming (ESOP), 2000, pp. 366–381.

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

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

[11] C. Hawblitzel, Adding operating system structure to language-based protection, Ph.D. thesis (June 2000).

[12] E. D. Berger, B. G. Zorn, K. S. McKinley, Reconsidering custom memory allocation, in: Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002, pp. 1–12.

[13] M. Hind, Pointer analysis: Haven't we solved this problem yet?, in: Proc. ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, UT, 2001, pp. 54–61.

[14] D. Walker, K. Watkins, On regions and linear types, in: Proc. ACM International Conference on Functional Programming (ICFP), 2001, pp. 181–192.

[15] L. P. Deutsch, D. G. Bobrow, An efficient, incremental, automatic garbage collector, Commun. ACM 19 (9) (1976) 522–526.

[16] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2001, pp. 59–69.

[17] D. Grossman, Safe programming at the C level of abstraction, Ph.D. thesis, Cornell University (2003).

[18] D. Grossman, Quantified types in imperative languages, ACM Transactions on Programming Languages and SystemsTo appear: accepted for publication, April 2005.

[19] L. Cardelli, S. Martini, J. C. Mitchell, A. Scedrov, An extension of system F with subtyping, in: Proc. International Conference on Theoretical Aspects of Computer Software, 1991, pp. 750–770.

[20] H. Xi, F. Pfenning, Dependent types in practical programming, in: Proc. ACM Symposium on Principles of Programming Languages (POPL), 1999, pp. 214–227.

[21] D. Mazières, A toolkit for user-level file systems, in: Proc. USENIX Annual Technical Conference, 2001, pp. 261–274.

[22] N. Provos, libevent — an event notification library, `http://www.monkey.org/~provos/libevent/`.

[23] M. Hicks, A. Nagajaran, R. van Renesse, MediaNet: User-defined adaptive scheduling for streaming data, in: Proc. IEEE Conference on Open Architectures and Network Programming (OPENARCH), 2003, pp. 87–96.

[24] M. Fluet, D. Wang, Implementation and performance evaluation of a safe runtime system in Cyclone, in: Informal Proceedings of the SPACE 2004 Workshop, 2004.

[25] D. Wang, A. Appel, Type-preserving garbage collectors, in: Proc. ACM Symposium on Principles of Programming Languages (POPL), 2001, pp. 166–178.

[26] Cyclone, version 0.9, `http://www.eecs.harvard.edu/~greg/cyclone/` (September 2005).

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

[28] M. Hertz, E. Berger, Quantifying the performance of garbage collection vs. explicit memory management, in: Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005.

[29] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, Programming with regions in the ML Kit (for version 4), Tech. rep., IT University of Copenhagen (Apr. 2002).

[30] A. Aiken, M. Fähndrich, R. Levien, Better static memory management: Improving region-based analysis of higher-order languages, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 1995, pp. 174–185.

[31] N. Hallenberg, M. Elsman, M. Tofte, Combining region inference and garbage collection, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2002, pp. 141–152.

[32] D. Gay, A. Aiken, Language support for regions, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2001, pp. 70–80.

[33] T. Terauchi, A. Aiken, Memory management with use-counted regions, Tech. Rep. UCB//CSD-04-1314, University of California, Berkeley (Mar. 2004).

[34] D. Walker, G. Morrisett, Alias types for recursive data structures, in: Proc. Workshop on Types in Compilation (TIC), 2000, pp. 177–206.

[35] F. Henglein, H. Makholm, H. Niss, A direct approach to control-flow sensitive region-based memory management, in: Proc. Principles and Practice of Declarative Programming (PPDP), 2001, pp. 175–186.

[36] P. Achten, R. Plasmeijer, The ins and outs of Clean I/O, Journal of Functional Programming 5 (1) (1995) 81–110.

[37] M. Fähndrich, R. DeLine, Adoption and focus: Practical linear types for imperative programming, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2002, pp. 13–24.

[38] A. Aiken, J. S. Foster, J. Kodumal, T. Terauchi, Checking and inferring local non-aliasing, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2003, pp. 129–140.

[39] C. Boyapati, A. Sălcianu, W. Beebee, M. Rinard, Ownership types for safe region-based memory management in real-time Java, in: Proc. ACM Conference on Programming Language Design and Implementation (PLDI), 2003, pp. 324–337.

[40] G. Bellella (Ed.), The Real-Time Specification for Java, Addison-Wesley, 2000.

[41] H. Baker, Lively linear LISP—look ma, no garbage, ACM SIGPLAN Notices 27 (8) (1992) 89–98.

[42] D. Harms, B. Weide, Copying and swapping: Influences on the design of reusable software components, IEEE Transactions on Software Engineering 17 (5) (1991) 424–435.