Experience With Safe Manual Memory-Management in Cyclone

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Dan Grossman
University of Washington
djg@cs.washington.edu

Trevor Jim
AT&T Labs Research
trevor@research.att.com

ABSTRACT

The goal of the Cyclone project is to investigate type safety for low-level languages such as C. Our most difficult challenge has been providing programmers control over memory management while retaining type safety. This paper reports on our experience trying to integrate and effectively use two previously proposed, type-safe memory management mechanisms: statically-scoped regions and unique 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 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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management, polymorphism*

General Terms

Languages, Experimentation, Reliability

Keywords

memory management, unique pointers, memory safety, regions, Cyclone

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 type-safety for C has led to many failures and security problems. The goal of our research is try to bring the "Mohammad of type safety" to the "mountain of existing C code."

Toward that end, we have been developing Cyclone, a type-safe dialect of C [23]. Cyclone uses a combination of programmer-supplied annotations, an advanced type system, a flow analysis, and run-time checks to ensure that programs are type safe. When we started the project, we relied entirely on heap allocation and the Boehm-Demers-Weiser (BDW) conservative garbage collector (GC) to recycle memory safely. BDW provides convenient interoperability with legacy libraries and makes it easy to support polymorphism without needing run-time type tags.

While the BDW collector provides convenience, it does not always provide the performance or control needed by low-level systems applications. In previous work, we described an integration of BDW with type-safe stack allocation and LIFO arena allocation. A region-based, type-and-effect system based upon the work of Tofte and Talpin [31] 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 [15]. We found that LIFO arenas were 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 server and event loops.

Since then, we have explored the integration of *unique pointers* into our memory management framework. Our unique pointers are closely related to typing mechanisms suggested by other researchers, including linear types [32], ownership types [9], alias types [28], and capability types [33]. The critical idea with all of these proposals is to make it easy to track locally the state of an object by forbidding uncontrolled aliasing. In Cyclone, a value with a unique-pointer type is guaranteed to be the only (usable) reference to an object. Such objects can be deallocated by the programmer at any time, and a modular flow analysis is used to ensure that the dangling pointer cannot be dereferenced in the rest of the computation.

Unique pointers are not a novel idea, but we found many challenges to implementing them in a full-scale safe language, as they interact poorly 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 of these features in a full-scale language implementation.

On the other hand, we found great synergies in the combina-

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

tion of uniqueness and regions. In particular, we were able to use the LIFO region machinery to support a form of "borrowed" pointers [8], which goes a long way in relaxing the burdens of uniqueness. We were also able to use unique pointers as capabilities for building further memory-management abstractions. In particular, we used unique pointers to control access to a form of dynamically-scoped arenas [18], and for building reference-counted objects and arenas.

In this paper, we describe our support for unique pointers and the extensions they enable; the Cyclone manual has further detail [10]. We then discuss our experience using these facilities to build or port a few target applications, including a multimedia overlay network, a small web server, a Scheme interpreter, an ftp server, and an image-manipulation program. Most of the applications were chosen because they are structured as (infinite) loops with loop-carried state and are thus not well-suited for LIFO arenas. Furthermore, we feel that these applications are representative for resource-limited platforms, such as cell-phones or embedded systems, where space is at a premium. 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 at least one application an improvement in throughput.

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

1. We show that the addition of unique pointers to a region-based language provides a flexible basis for building type-safe, manual memory management, which can complement or replace GC.

2. We confirm that the resource requirements of some important applications can be significantly improved through type-safe, manual memory management.

2. REGIONS IN CYCLONE

All of the memory management facilities in Cyclone revolve around *regions*. 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 another region, whose objects are individually deallocated by the collector.

Each region in a program is given a compile-time name, either explicitly by the programmer or implicitly by the compiler. For example, the heap region's name is `‘H` and the region name for a function `foo`'s stack frame 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*‘r`.

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 using a type-and-effects system in the style of Tofte and Talpin [31].

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 live in distinct regions, and that all of these regions are live on input. A unification-based algorithm is used to infer instantiations for region-polymorphic functions and the regions for local variables. This drastically cuts the number of region annotations needed for programs. 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

The basic region system we have described easily supports a form of *arenas* that have stack-like last-in-first-out (LIFO) lifetimes, but also support dynamic allocation.[1] 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`. Calling the primitive `rmalloc(h,...)` allocates space within region `‘r`. When the declaring lexical scope concludes, the handle and the contents of the arena are deallocated.

```
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_huffman_decode(infile, huffman_tree,
                          symbol_stream, ...);
} else ...
```

Figure 1: LIFO Arena example

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* image compression/decompression benchmark in Section 4). If the image in the specified `infile` is compressed using huffman encoding, then the `read_huffman_tree` routine deserializes the huffman tree from the file into a `code_node` tree allocated in region `‘r`. This tree is used to decompress the remaining file contents into the `symbol_stream` array, which is then further decompressed in the continuation. 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 the block. Obviously, statically allocating the space for the tree would be problematic since the size of the tree depends on the contents of `infile`.

Unfortunately, we found that the LIFO restriction on arena lifetimes was often too limiting. That is, we often wished that we could deallocate the 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 can be carried from one iteration to the next, unless they are copied to an arena declared outside the loop. But then all of the data placed in an outer arena would persist until the entire loop terminates. For loops that do not terminate, such as a server request loop or event loop, this is a disaster as the LIFO restriction can lead to unbounded storage requirements.

3. UNIQUE POINTERS

Very often the limitations of stack allocation and LIFO arenas can simply and conveniently be overcome by using garbage collection (GC). However, GC may not always be compatible with the

---

[1] Our previous paper [15] referred to LIFO arenas as *dynamic regions* due to their dynamically-determined 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.

performance needs of an application. For example, embedded systems and network servers sometimes require bounds on space consumption, pause times, or throughput that may be hard to achieve with GC. Therefore, we decided to extend Cyclone with a suite of mechanisms that would permit manual object deallocation without imposing a LIFO restriction. We emphasize that 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 very precise information regarding which pointers may alias other pointers. Though there are impressive analyses that compute such aliasing information, they usually require whole-program analysis to achieve any level of accuracy.

An alternative solution that has been proposed many times is to restrict or avoid aliasing altogether, so that reasoning about type-states can be done locally. One extreme is to require that pointers which are deallocated be *unique* (i.e., have no aliases.) In what follows, we briefly describe how we have incorporated unique pointers into Cyclone, enabling support for per-object deallocation, reference-counting, and arenas with dynamic lifetimes.

To distinguish a unique pointer from a heap-, stack-, or arena-allocated pointer, we use a special region name `‘U`. A pointer value with type `T*‘U` is created by calling `malloc` and can be deallocated via `free` at any time.

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 attempt to use 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 (e.g., in an assignment or function call) 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;         // q not consumed
free(q);        // consumes q
*p = *q;        // illegal: p & q consumed
```

The first assignment aliases and consumes `p`, while the call to `free` consumes `q`. Therefore, attempts to dereference `p` or `q` are illegal. Dereferencing a unique pointer does not consume it since it does not copy the pointer, as the first dereference of `q` shows.

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. For instance, 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, we issue a warning that `p` might leak, since the type-states do not match. Fortunately, we can link in the garbage collector to ensure the object is reclaimed. We considered making this an error as in Vault [11], but found that exception handlers and shared unique pointers (described below) generated too many false alarms. Thus, we settle for a warning and rely upon the GC as a safety net.

We allow unique pointers to be placed in non-unique objects (e.g., a global variable or heap-allocated object) that might have multiple aliases. To ensure that unique pointers remain unique, we must somehow limit access through these paths. In most 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. In Cyclone, we have pointer types that do not admit `NULL` as a value, so destructive reads are not always an option. Therefore, we provide an explicit *swap* operation (“`:=:`”) that allows one to swap one unique object for another (including `NULL` where permitted.) 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.

Here is a simple example of the utility of swap:

```
int *‘U g = NULL;
void init(int x) {
  int *‘U temp = malloc(sizeof(int));
  *temp = x;
  g :=: temp;
  if (temp != NULL) free(temp);
}
```

Here, `g` is a global variable holding a unique pointer. The `init` routine creates the unique pointer and stores it in `temp`. Then, the value of the temporary is swapped for the value of `g`. Afterward, if `temp` is not `NULL`, then we free the pointer. 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. Atomic swap can be used to build useful concurrent protocols, while simple extensions, like compare-and-swap, can be used to build arbitrary wait-free structures [20].

In general, we can perform assignments directly without need of swapping as long as we assign a unique pointer to a *unique path*. A unique path u has the form

u `::= x |` u`.m |` u`->m | *`u

where `x` is a local variable and u is a unique pointer. Its syntactic difference with normal assignment expresses swap’s runtime cost.

Finally, we remark that placing unique pointers in non-unique objects can lead to subtle “leaks”. For instance, there is no guarantee that when a unique pointer is passed to `free` that the object to which it points does not contain live, unique pointers. Similarly, when we write a unique pointer into a shared object (e.g., a global) there is no guarantee that we are not overwriting another live unique pointer.

3.1 Borrowing Unique Pointers

Unique pointers make it easy to support explicit deallocation, but they often force awkward coding idioms just to maintain uniqueness. For example, we must forbid using pointer arithmetic on unique pointers, since doing so could allow the user to call `free` with a pointer into the middle, rather than the front, of an object, confusing the allocator. (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.) As another example, we often want to pass a copy of a pointer to a function that does not consume the pointer, and leave the original copy as unconsumed.

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 of the borrowed copies. In Cyclone, a pointer is borrowed using an explicit `alias` declaration, similar to Walker and Watkins’ `let-region` [35], and the LIFO region machinery prevents the borrowed pointer from escaping.

Consider the example in Figure 2 (adapted from the *Kiss-FFT* benchmark). The `do_fft` function allocates a unique pointer `x` to

```
fft_state *‘U fft_alloc(int n, int inv);
void fft(fft_state *‘r st,...);
void fft_free(fft_state *‘U st);
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);
}
```

Figure 2: Pointer borrowing example

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`. Within the scope of the `alias` declaration (which is the entirety of the `for` loop), we may freely copy `a` and pass it to functions, just as if it were allocated on the stack, in the heap, or in an arena. However, throughout the scope of the `alias`, the original unique pointer `x` is considered to be consumed. This prevents the object to which it refers from being deallocated. At the end of the block, any copies of `x` will be treated as 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.

Using a fresh, lexically-scoped region name is crucial to the soundness of the `alias` construct. For example, suppose an incorrect program attempts to assign a borrowed pointer of type `int*‘s` to a global variable (thus permitting the pointer to escape.) Such a program cannot type-check because the global cannot have a compatible type since `‘s` is not in scope.

Finally, it is important to note that we provide a limited form of `alias`-inference around function calls to simplify programming and cut down on annotations. In particular, whenever a region-polymorphic function (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 arguments to be freely duplicated within the callee. As a result, we can rewrite `do_fft` to be:

```
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` around the function call to `fft`.

3.2 Polymorphism

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

Aliasability	May have aliases	May create aliases
normal	×	×
unique (U)		
top (T)	×	

$$\text{normal} \leq \text{top (T)}$$
$$\text{unique (U)} \leq \text{top (T)}$$

Table 1: Kinds and Aliasabilities

```
‘a copy(‘a x,‘a *‘U y) {
  *y = x;
  return x;
}
```

The syntax `‘a` denotes a Cyclone *type variable* which stands for an arbitrary type. The `copy` function copies `x` into the pointer argument `y` and returns `x` as the result, effectively creating two copies of `x`. Consider if we call `copy` with a unique pointer:

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

When calling `copy`, `x` is consumed, but two copies of it are made available to the caller. Thus the caller can free one of them and then erroneously use the other. To prevent this situation, we need to distinguish between polymorphic regions and values that can be copied (i.e., in the body of the `copy` function) and those that cannot.

We classify types with both an *aliasability* and a *kind*. Kinds describe the structure of types, e.g., `B` describes boxed (i.e., pointer) types, `R` describes regions, etc. Aliasabilities indicate how a type may be aliased, and are shown in Table 1. A *normal* aliasability indicates that a type’s objects may have aliases and that aliases can be freely created. In contrast, a *unique* aliasability indicates that a type’s objects are not aliased, nor can they ever be. For example, the region `‘U` has kind `UR` and the type `int *‘U` has kind `UB`. Normal aliasability is the default. As such, to prohibit calling `copy` with a unique pointer, its fully annotated prototype should be

```
‘a::B copy(‘a x,‘a *‘U y);
```

This states that `‘a` has boxed kind `B`, and normal aliasability. The call to `copy` in the above example would be disallowed because `x` has type `int *‘U`, which has kind `UB`, and `UB` $\not\leq$ `B`. Note that `‘a` has boxed kind by default and need not be annotated explicitly as we have done here.

For more code reuse, we define a third aliasability `T` (for “top”), whose types’ objects could have aliases, but for which no new aliases may be created. For instance, a value of type `‘a::TB` cannot be freely duplicated *and* cannot be considered unique. These conditions permit using unique- or normal-kinded types where we expect top-kinded types, creating the subaliasing relationship shown in Table 1. This enables writing functions polymorphic over aliasability.

3.3 Reference Counting

Even with borrowing and polymorphism support, unique pointers can only be used 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.

Region Variety	Allocation	Deallocation		Aliasing
	(objects)	(what)	(when)	(objects)
Stack	static	whole region	exit of lexical scope	unrestricted
LIFO Arena	dynamic			
Dynamic Arena			manual	
Heap (`‘H`)		single objects	automatic (GC)	
Unique (`‘U`)			manual	restricted
Reference-counted (`‘RC`)				

Table 2: Summary of Cyclone Regions

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 inherited for reference-counted objects.

We define a new *reference-counted region* `‘RC`, whose objects, when allocated, are prepended with a hidden reference-count field. As with unique pointers, the flow analysis prevents the user from making implicit aliases. Instead, `‘RC` pointers must be copied *explicitly* by using the `alias_refptr` function, which increments the reference count and returns a new alias, without consuming the original pointer. In essence, both of the `‘RC` values 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; if the count becomes zero, the memory is freed.

```
struct conn *‘RC cmd_pasv(struct conn *‘RC 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);
  return c;
}
```

Figure 3: Reference counting example

Consider the example in Figure 3, adapted from the *BetaFTPD* benchmark. In BetaFTPD, `conn` structures and `ftran` structures mutually refer to one another. 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 `‘RC` pointers as if they were unique forces programmers to manipulate reference counts explicitly. Instead of `free`, one uses `drop_refptr`, which calls `free` if the reference count becomes zero. Otherwise, the compile-time restrictions on reference-counted pointers are like those for unique pointers. For example, if `y` holds a reference-counted pointer, the assignment `x=y` consumes `y`. So there is at most one usable alias *per call* to `alias_refptr`.

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. Thus, the programmer has complete control over where reference counts are manipulated.

3.4 Dynamic Arenas

Coming full circle, 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 having 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 represented as a unique pointer and is consumed when the arena is destroyed, 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;
}
```

Figure 4: Dynamic Arena example

Consider the example in Figure 4, adapted from the Cyclone compiler's lexer: The function `insert_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` is not accessible, so `t` cannot be dereferenced. (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.) Within the function, the arena is opened via the syntax `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.

Program	Description	Non-comment Lines of Code			Manual
		C	Cyc	Cyc (+manual)	mechs
Boa	web server	5217	± 284 (5%)	± 91 (1%)	U(D)
BetaFTPD	ftp server	1146	± 191 (16%)	± 225 (21%)	UR
Epic	image compression utility	2123	± 217 (10%)	± 114 (5%)	UL
Kiss-FFT	fast Fourier transform routine	453	± 73 (16%)	± 20 (4%)	U
MediaNet	streaming data 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

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, where `‘RC` pointers are used 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.

Thus, the addition of unique pointers to our region framework gives us a number of memory-management options, which are summarized in Table 2 . It illustrates the flexibility that programmers have, particularly in choosing how to deallocate objects. As the alias restrictions indicate, a particular choice essentially trades ease of use for better control over object lifetime. In the following section, we describe our experience trying to use these facilities in various applications.

4. APPLICATIONS

Table 3 describes the benchmark programs with which we experimented. For programs we ported from C (Boa, BetaFTPD, Epic, and Kiss-FFT) it shows the non-comment lines of code of the original program, and then the changes due to porting to Cyclone to use the GC, and then the additional changes needed to use manual mechanisms. 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 eliminated the need for GC entirely. We describe the program and the coding process below. Performance experiments for these programs are presented in the next Section.

4.1 Porting Experience

The process of porting from C to Cyclone is made easiest by placing all dynamically-allocated data in the heap region and letting BDW take care of recycling the data. Most of the changes involve differences between C and Cyclone that are not memory-management related, such as introducing *fat* pointer annotations. (Fat pointers carry run-time array-bounds information with them used 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 between those data whose lifetimes are scoped from those whose are not. 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 unique region, and use the `alias` construct to allow temporary aliasing until they can be freed. For example, in both Epic and Kiss-FFT, we merely had to change a declaration from something like

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

to instead be

```
T*‘U q_pyr = malloc(...);
```

All `alias` statements were inferred automatically when calling subroutines that wished to alias the array (see Figure 2). 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 5. 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 is moved between a blocked queue and a ready queue, and when complete, the request is moved 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 complete. We also experimented with a version that used dynamic arenas for requests instead of the custom allocator, but found that this adversely

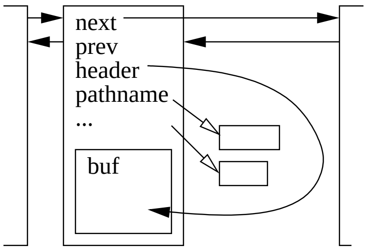


Figure 5: Request data structure in Boa

affected throughput (we are still investigating why). We remark that this alternative required changes only to the topmost request management routines; the internal request processing routines could remain the same.

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 can be seen from Table 3, this required 21% of the code to be changed, significantly more than the other ports. 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 (see Figure 3). 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. Thus, our experience with reference counting was mixed.

Because the changes to legacy code do not change the underlying data structures or logic of the application, we have not found the introduction of application-level bugs a major problem. Straightforward testing, using the original C program as the ground truth, appears suitable in practice, though the inherent "danger" of changing existing code cannot be completely eliminated.

4.2 Cyclone Applications

In addition to porting C programs, we have written three Cyclone programs from scratch that use our manual mechanisms. Here we describe which mechanisms we used and why.

4.2.1 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 [25] and libevent [27]. 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 `alias`s.

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.

4.2.2 MediaNet

MediaNet [22] 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.

We use a data structure called a *streambuff* for each packet, similar to a Linux `skbuff`:

```
struct StreamBuff { <i::I>
  ... // three omitted header fields
  tag_t<i> numbufs;
  struct DataBuff<`RC> bufs[numbufs];
};
```

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`), 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 [37]. *Databuffs* store packet data:

```
struct DataBuff {
  unsigned int ofs;
  byte ?`RC buf;
};
```

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 unique region. 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 6. 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.

MediaNet's DAG of operations is stored in a dynamic arena. We found it convenient to allocate some objects in the heap, such as the objects representing connections. This means that we must use the GC to manage these objects. An earlier version of MediaNet stored all packet data in the heap as well, 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

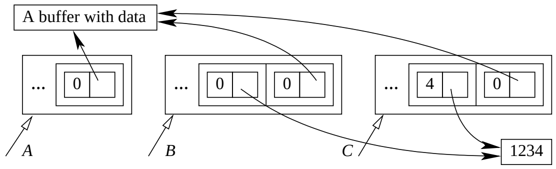


Figure 6: Pointer graph for three streambuffs

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 a couple of utility files. Out of nearly 9000 non-comment lines, we used swap 74 times and `alias` 67 times, of which 67% were automatically inferred.

4.2.3 CycScheme

Using a combination of our dynamic arenas and unique pointers, Fluet and Wang [13] 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 [36], written entirely in type-safe Cyclone. All data from the interpreted program are allocated in an 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. We simply observe that our system of memory management was flexible enough for this interesting application. In particular, this shows that at least in principle, it is possible for Cyclone programs to code up their own copying garbage collectors tailored for a particular application. However, we remark that the approach is limited to straightforward copying collection and does not accommodate other techniques, including generations or mark/sweep.

5. PERFORMANCE EXPERIMENTS

To understand the benefit of our proposed mechanisms, we compared the performance of the GC-only versions of our sample applications to the ones using manual mechanisms. Our measurements exhibit two trends. First, we found that elapsed time is similar for the GC and manual versions of the programs. Indeed all of 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 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 the Appendix. 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 0.8 which, along with the benchmarks, is publicly available [10]. By default, Cyclone programs use the Boehm-Demers-Weiser (BDW) conservative collector [6], version $6.2\alpha4$, 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. When programs do not need GC, they are compiled with the Lea general-purpose allocator [24], version 2.7.2. Cyclone compiles to C and then uses `gcc` version 3.2.2, with optimization level `-O2`, to create executables.

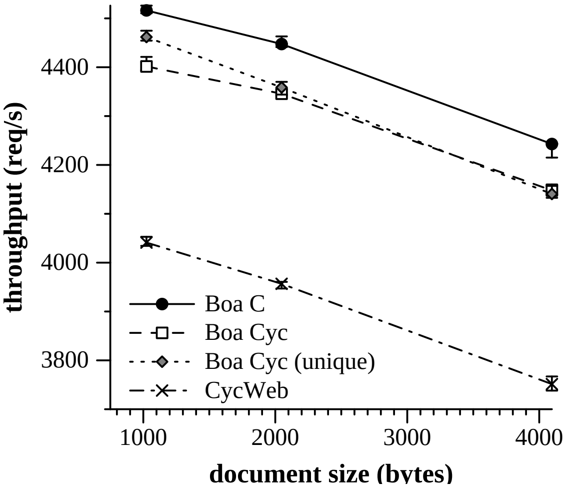


Figure 7: Throughput for CycWeb and Boa

5.1 Boa and CycWeb

We measured web server throughput using the SIEGE web benchmarking tool[2] 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 7—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, and the error bars show the quartiles. For Boa, the performance difference between the C and Cyclone versions is between 2 and 3%, and the differences between the various Cyclone versions are 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 comes within 10–15% of Boa in C.

Avoiding GC has greater benefit when considering memory footprint. Figure 8 depicts three traces of Boa's memory usage while it serves 4 KB pages to 6 concurrent users. The first trace uses GC, while the second two make use of unique pointers. The second (unique+GC) uses BDW as its allocator (thus preventing inadvertent memory leaks), while the third uses the Lea allocator.[3] The x-axis is elapsed time, while the y-axis plots memory consumed. The graph shows memory used by the heap region and by the unique region, 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 (each GC,

---

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

[3] The throughput of both versions is essentially the same, so only one line is shown in Figure 7.

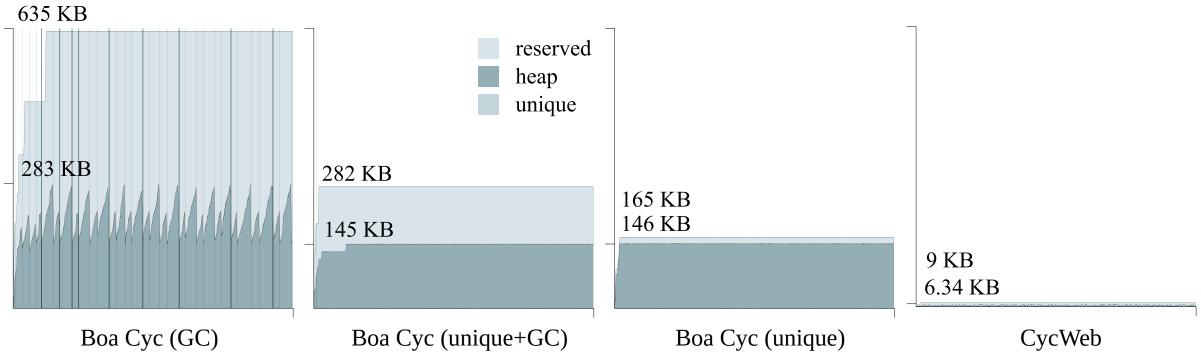


Figure 8: Memory footprint of Cyc Boa versions

of which there are a total of 33 in this case, is depicted as a vertical line). To ensure reasonable performance, the collector reserves a fair amount of headroom from the OS: 635 KB in this case. 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 heap sizes and levels of concurrent access and found the trends to be similar. Very little data is allocated in the unique region (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 of 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 [26] 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.

Figure 9 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 error bars plot the quartiles. The two configurations perform roughly the same for the smallest packet sizes, but *GC only* quickly falls behind as packets reach 256 bytes. Both curves level off at 4 KB packets, with the *GC+free* case achieving 23% better throughput. The odd leveling of the slope in the *GC only* curve at 2 KB packets results in a 70% difference.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.

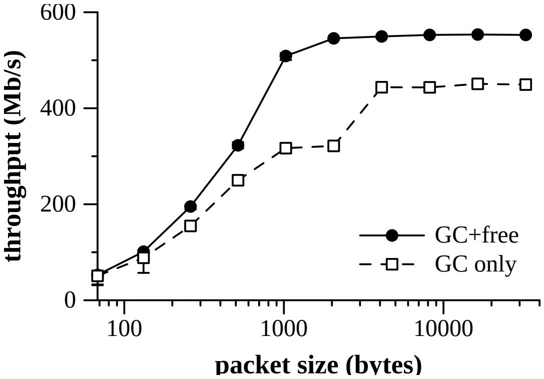
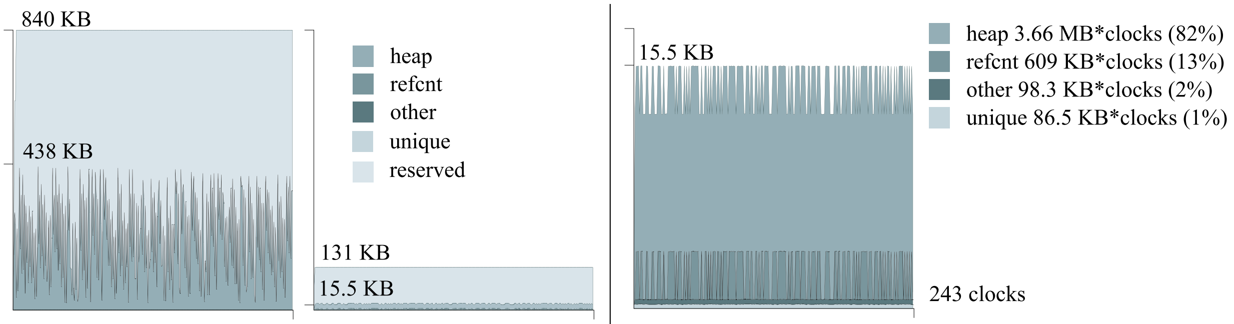


Figure 9: MediaNet throughput

Figure 10 illustrates the memory usage of MediaNet when forwarding 50,000 4 KB packets. This graph has the same format as the graph in Figure 8; it shows the heap, unique, and reference-counted regions, and the dynamic region in which the configuration is stored (labeled "other"). The *GC only* configuration stores all data in the heap region, which exhibits a sawtooth pattern with each peak roughly coinciding with a garbage collection (there were 553 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 15.5 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 heap region (the reserved space is not shown), while the feathery band below it is the reference-counted region. Below that is the dynamic region and finally the unique region.

These performance trends are consistent with other studies comparing GC and manual memory management [38, 21]. What we have shown is that some simple and safe manual mechanisms can *complement* GC in attacking problems of memory management. They give programmers more control over the performance of their pro-



MediaNet (GC only) MediaNet (GC+free) MediaNet (GC+free) close up

Figure 10: MediaNet memory utilization

grams, without undue programming burden, and without need to compromise safety.

6. RELATED WORK

The ML Kit [30] implements Standard ML with (LIFO) arenas. Type inference is used to automatically allocate data into arenas. Various extensions have relaxed some LIFO restrictions [2, 16], but unique pointers have not been considered.

The RC language and compiler [14] 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 type-safety guarantees of Cyclone.

Use-counted regions [29] 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 [32], alias types [28, 34], linear regions [35, 19], and uniqueness types [1] 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 from the Vault project [11, 12] 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 Foster and Aiken’s `restrict` [3]) 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 more user annotations (or a global alias analysis.)

Unique pointers and related restrictions on aliasing have received considerable attention as extensions to object-oriented languages. Clarke and Wrigstad [9] 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 [7], which uses regions to avoid some run-time errors in Real-Time Java programs [5]. 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 [4, 17]. Alias burying [8] 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.

7. CONCLUSIONS

Cyclone now supports a rich set of safe memory-management idioms beyond garbage collection:

- *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.
- *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.

8. REFERENCES

[1] Peter Achten and Rinus Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] Alex Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, 1995.

[3] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 129–140, June 2003.

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

[5] Gregory Bellella, editor. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

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

[7] Chandrasekhar Boyapati, Alexandru Sălcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, June 2003.

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

[9] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, July 2003.

[10] *Cyclone, version 0.8*. Available at `http://www.eecs.harvard.edu/~greg/cyclone/`.

[11] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, June 2001.

[12] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, June 2002.

[13] Matthew Fluet and Daniel Wang. Implementation and performance evaluation of a safe runtime system in Cyclone. In *Informal Proceedings of the SPACE 2004 Workshop*, January 2004.

[14] David Gay and Alex Aiken. Language support for regions. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 70–80, June 2001.

[15] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, June 2002.

[16] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 141–152, June 2002.

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

[18] Chris Hawblitzel. *Adding Operating System Structure to Language-Based Protection*. PhD thesis, June 2000.

[19] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proc. Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, September 2001.

[20] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[21] Matthew Hertz and Emery Berger. Automatic vs. explicit memory management: Settling the performance debate. Technical Report CS TR-04-17, University of Massachussetts Department of Computer Science, 2004.

[22] Michael Hicks, Adithya Nagajaran, and Robbert van Renesse. MediaNet: User-defined adaptive scheduling for streaming data. In *Proc. IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, pages 87–96, April 2003.

[23] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, pages 275–288, June 2002.

[24] Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[25] David Mazières. A toolkit for user-level file systems. In *Proc. USENIX Annual Technical Conference*, pages 261–274, June 2001.

[26] Mike Muuss. The story of TTCP. `http://ftp.arl.mil/~mike/ttcp.html`.

[27] Niels Provos. libevent — an event notification library. `http://www.monkey.org/~provos/libevent/`.

[28] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. European Symposium on Programming (ESOP)*, pages 366–381, March 2000.

[29] Tachio Terauchi and Alex Aiken. Memory management with use-counted regions. Technical Report UCB//CSD-04-1314, University of California, Berkeley, March 2004.

[30] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, September 2001.

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

[32] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, April 1990. IFIP TC 2 Working Conference.

[33] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 24(4):701–771, July 2000.

[34] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proc. Workshop on Types in Compilation (TIC)*, pages 177–206, September 2000.

[35] David Walker and Kevin Watkins. On regions and linear types. In *Proc. ACM International Conference on Functional Programming (ICFP)*, pages 181–192, September 2001.

[36] Daniel Wang and Andrew Appel. Type-preserving garbage collectors. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–178, January 2001.

[37] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. pages 214–227, January 1999.

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

Test	C time(s)	Cyclone time (GC)			
		checked(s)	factor	unchecked(s)	factor
Epic	1.06 ± 0.00	1.60 ± 0.00	1.51×	1.05 ± 0.01	0.99×
Kiss-FFT	1.33 ± 0.00	3.21 ± 0.01	2.41×	1.30 ± 0.01	0.98×
BetaFTPD	2.17 ± 0.02	2.25 ± 0.02	1.04×	2.22 ± 0.01	1.02×
		Cyclone time (+manual)			
Epic	1.06 ± 0.00	1.61 ± 0.01	1.52×	1.06 ± 0.00	1.00×
Kiss-FFT	1.33 ± 0.00	3.22 ± 0.01	2.42×	1.31 ± 0.00	0.98×
BetaFTPD	2.17 ± 0.02	2.24 ± 0.01	1.03×	2.23 ± 0.02	1.03×

Table 4: Benchmark performance

APPENDIX

A. ADDITIONAL PERFORMANCE MEASUREMENTS

This section presents performance measurements for the benchmarks not considered in Section 5. In general, these benchmarks exhibit the following trends (as mentioned in the body of the paper):

- Using manual memory management mechanisms does not improve the execution time of the program relative to GC.
- Using manual memory management does allow the memory footprint of the program to be reduced.

We measured the performance of each program as follows. For Epic, we used it to compress and decompress a large image file. For Kiss-FFT, we performed 1024 size 10000 FFT's, using the benchmark program provided with the distribution. For BetaFTPD, we used WGET[4], a utility for retrieving files from HTTP and FTP servers, to retrieve a 20 KB file via anonymous FTP 1000 times, piping it to `/dev/null`.

A.1 Elapsed Time Measurements

The results of measuring the elapsed time of each benchmark are shown in Table 4. Here we measure the C and Cyclone versions, with the top three rows considering Cyclone using GC, and the bottom three using manual mechanisms (no GC needed). We also report the performance of Cyclone with and without array bounds checks enabled. Each number reports elapsed time in seconds, and is the median of 21 trials, with ± referring to the scaled semi-interquartile range (SIQR). The SIQR measures variance, similar to standard deviation, by calculating the distance between the quartiles and scaling it to the median.

For the computationally-intensive Epic and Kiss-FFT programs, we see that Cyclone can be substantially slower than C due to array bounds checks. While Cyclone does some array-bounds check elimination, this is an area of current work in the compiler. With these particular benchmarks, the problem is the use of pointer arithmetic. Our compiler could eliminate many more checks if we were to restructure the program to use array indexes instead.

For BetaFTPD there is no appreciable difference between the C and Cyclone versions. To acquire a file via anonymous FTP requires roughly six configuration commands, at which time the client instructs the server to connect back to it on a specified port to send the data. As a result, retrieving a file using anonymous FTP is very much I/O-bound, and quite time-consuming, so there is little concern about the CPU-time or pause-time overhead incurred by garbage collection (or reference-counting, for that matter). On the other hand, BetaFTPD is clearly not well optimized, as each FTP takes 22 ms.

Of most concern to the topic of this paper, we can see that using manual memory management (in this case, unique and reference-counted pointers) did not provide a performance advantage relative to Cyclone using GC when considering elapsed time.

A.2 Memory Footprint Measurements

Test	KB Footprint (GC)		KB Footprint (+manual)		
	data	resv	data	resv (BDW)	resv (Lea)
Epic	17475	23400	13107	15585	13128
Kiss-fft	400	725	400	725	402
BetaFTPD	183	356	3.3	65	8

Table 5: Benchmark Memory Footprint

Statistics for memory footprint are shown in Table 5. For each benchmark we report the peak memory usage for the data memory and reserved memory, in kilobytes. The first group of numbers are for the GC case, while the last group are for the manual case, and we consider the reserved memory for the case when using the BDW collector as the allocator or the Lea allocator. None of the manual versions of these programs require garbage collection.

For Epic there is a memory utilization advantage to the manual case because we are able to free some data early, i.e., during the compression process. On the other hand, the FFT program is set up to only free its memory upon conclusion, so there is no real effect on data footprint, and thus the only benefit is to reduce the about of reserved space by linking in the Lea allocator. The trend for BetaFTPD is similar to that of Boa, shown earlier. In particular, when using garbage collection both the data footprint and the reserved memory required are much higher than for the manual mechanisms. We do not show the memory-consumption graphs here, but they are essentially the same as Boa (and MediaNet): a sawtooth pattern for the GC case (for a total of 11 GCs during the run), and a smooth trend for the manual case.

---

[4] `http://www.gnu.org/directory/wget.html`