Short Presentation: Combining Garbage Collection and Safe Manual Memory Management

Michael Hicks
University of Maryland
mwh@cs.umd.edu

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

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

1. INTRODUCTION

Garbage collection (GC) provides an elegant, convenient, and safe approach to managing memory. For many applications, it is an appropriate technique for all data. For other applications, it works well only for most data. For example, it may complicate data-level interoperability with legacy code or exhibit poor performance with respect to a few critical data types. In other systems, such as embedded devices and operating systems, the environment may simply prove too hostile for GC to be a realistic technique.

Unfortunately, an approach to memory management is often a system-wide decision, made when choosing a run-time system or even when choosing a language. Furthermore, using `malloc` and `free`, as in C, usually requires sacrificing safety. We believe programming languages should support manual and automatic memory-management, allowing programmers to choose the mechanisms most appropriate for (different parts of) their applications. They should us GC when *possible* and manual techniques when *necessary*, with smooth intra-language evoluation paths among the options.

At the same time, the language should remain *safe*, which to us means (at least) two things. First, programs should not crash (or worse, silently misbehave) because the same memory is simultaneously used for different types. Second, parts of an application should be able to maintain private state by allocating fresh memory and restricting access to it. The key to both requirements is to prohibit dangling-pointer dereferences; making the allocator type-safe by partitioning memory into typed pools satisfies the first requirement but not the second.

Toward this end of safe, flexible memory management we have designed and implemented a comprehensive memory-management framework meeting our requirements in the context of the Cyclone programming language. Cyclone is a type-safe, C-like language intended for use in systems programs where control is needed over low-level details such as data representations and resource management. In the remainder of this paper, we provide an overview of our approach in Cyclone, summarizing technical results presented in other papers [2, 3]. We present some performance measurements that show how manual and automatic memory management can be combined to good effect, and lay out plans for future work.

2. MEMORY MANAGEMENT IN CYCLONE

The unifying theme of Cyclone's safe memory management support is the idea of a *region*. We use the term loosely: when we say an object "lives" in a certain region, we are really saying that the object has certain lifetime characteristics, which are related in some way to the lifetimes of other objects in the same region. Pointer types are annotated (often implicitly) with the region of their pointed-to object: $\tau * \rho$ is the type of a pointer to data of type τ, "located" in region ρ.

Cyclone has different varieties of regions, summarized in Table 1. The first three rows describe regions whose objects are deallocated all at once, when the entire region is deallocated. *Stack* frames are just as in C: the number of objects allocated is statically determined, and the frame is deallocated at the end of the frame's scope. *Lexical* regions generalize stack frames by allowing objects to be dynamically allocated (similar to, but more general than C's `alloca`). *Dynamic* regions allow whole-region deallocation to occur at any time, not just at the end of the region's lexical scope. However, a special `open` construct must be used to access the region, and deallocation is forbidden while the region is opened (see below). Each stack frame and region gives rise to a different region name `‘r` usable in a pointer type of the form $\tau * $ `‘r`.

The remaining three rows describe regions that themselves "live forever," but whose objects can be individually deallocated at any time. Objects in the *heap* region, `‘H`, are deallocated by an automatic garbage collector. Objects in the *unique* region, `‘U`, can be deallocated manually, using `free`. Dangling pointers dereferences are prevented by a static type- and control-flow analysis that ensures that there is only one usable pointer to an object in `‘U` at any time, and that makes the argument of `free` unusable after `free` returns. A pointer into `‘U` can itself be stored in a region other than `‘U`, but such a pointer can only be accessed by atomically swapping it with the contents of a local variable, thus ensuring our uniqueness invariant. Swapping generalizes "destructive reads" (which always swap *null* into the container). Objects in the *reference-counting* region, `‘RC`, have a hidden count field maintained by the run-time system. Like unique pointers, pointers to reference-counted objects cannot be implicitly aliased. Instead, users call a run-time function that returns an explicit alias (and increments the count). A second function destroys a pointer and decrements the count, freeing the object when the count reaches zero.

Our prior publications described the type system for regions [2] and unique pointers [3] in detail; this paper focuses on how all of Cyclone's memory management options can be used together in real programs, synergistically, and without leading to excessive code duplication. Several features make

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

Table 1: Summary of Cyclone Regions

this possible.

First, an `alias` construct allows temporary (scoped) aliasing of a unique pointer and prohibits (temporarily) freeing the object. By using a new, scoped region name for the aliasable copy of the pointer, we leverage the region system and need not impose ad hoc restrictions to ensure the aliasable copy does not escape its scope. In fact, a notion of subtyping based on one region outliving another allows us to extend `alias` to deep read-only data structures, which makes it more powerful than any analogous construct we are aware of.

Second, unique pointers to region handles make our new dynamic regions a derived construct. Using `open` on such a handle is just like `alias` except that it also provides access to the corresponding region. Hence the same static machinery that prevents freeing temporarily aliased objects prvents freeing temporarily accessible regions. In fact, the less powerful regions in our previous work are equivalent to creating one of our new regions, immediatly opening the region, and deallocating the region as soon as control leaves the open construct. (In practice, we use a slightly faster implementation than this encoding.)

Third, polymorphism and kinding can avoid code duplication. Being polymorphic over the region name of pointer arguments is crucial for code taking pointers into regions created by callers since the code cannot know the name of the regions. More novel is our ability to write functions that can be called with pointers into regions or unique pointers. The key is to have a kind imposing the lifetime restrictions of both kinds of pointers: Code cannot alias such pointers *and* cannot deallocate them.

3. PRELIMINARY RESULTS

In addition to ensuring the safety guarantees described earlier, our approach to integrating manual memory management with GC must prove convenient and useful for performance. If our constructs do not provide enough added control for users to improve performance, they have little value. Similarly, if the constructs require unnatural coding styles, extensive annotations, or pervasive code duplication, they will be too burdensome. In this section, we outline our experience writing two applications, a webserver and a multi-media overlay network, providing encouraging qualitative and quantitative results.

3.1 Web server

We built a simple, space-conscious web server that supports concurrent connections using non-blocking I/O and an event library in the style of libasync [5] and libevent [6]. 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; Cyclone's existential types make callbacks straightforward. The event library uses polymorphism to allow callbacks and their environments objects to be allocated in arbitrary regions. 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. The entire web server, at about 800 lines of code, uses swap 16 times, and `alias` 5 times.

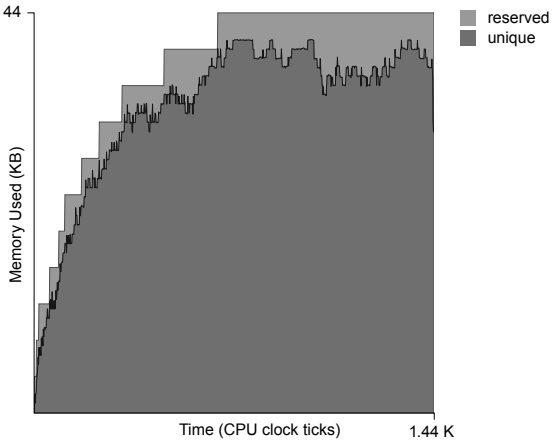

Figure 1: Memory use of the web server with up to 40 concurrent clients

Because concurrent HTTP connections overlap in a non-nested fashion and can have widely varying lifetimes, callbacks are not suited to allocation in lexical or dynamic regions. Rather than use garbage collection, we use unique pointers exclusively. When a client requests a file, the server allocates a small buffer for reading the file and sending it to the client in chunks. Callbacks are manually freed by the event loop when the callback is invoked (they must be reregistered if an entire transaction is not completed); each callback is responsible for freeing its own environment, if necessary.

Figure 1 shows that we achieved tight control over memory use. It shows the memory use of the web server under a sustained load with a maximum of 40 concurrent connections, using a buffer size of 1 KB. The x-axis plots CPU time in terms of clock ticks (as determined by the `clock()` system call), while the y-axis plots memory consumed. We also plot the total space the allocator acquires from the operating system.

Our profiler confirms that all dynamic memory is stored in

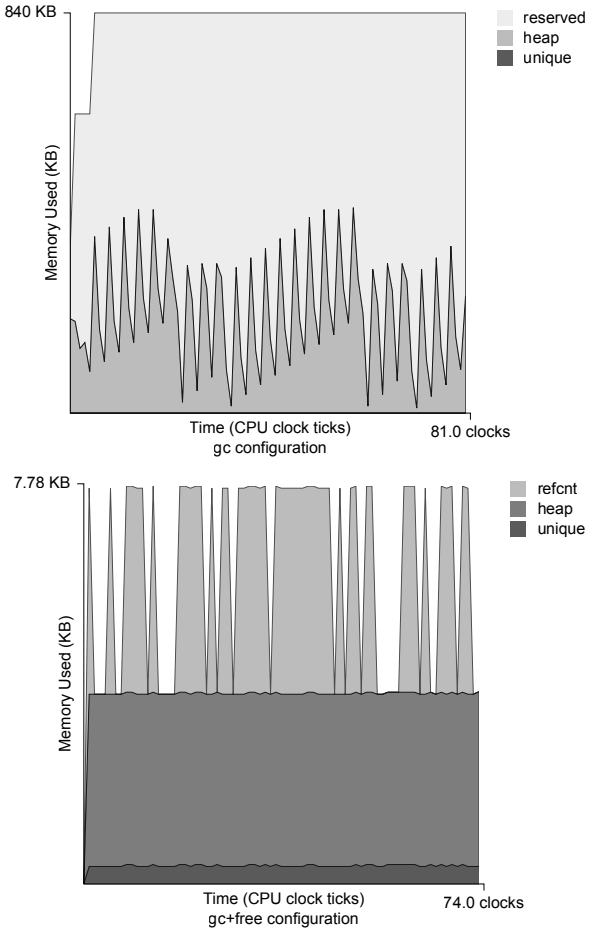

Figure 2: MediaNet memory profile (4 KB packets)

the unique region, which occupies at most 40KB (1KB per 40 connections) of the total reserved memory of 44KB. The server thus makes very efficient use of dynamic memory, with little fragmentation. And, of course, there are no pauses for garbage collection.

3.2 MediaNet

MediaNet [4] is an overlay network whose servers 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, meaning that region allocation is again inappropriate.

We use a datastructure called a *streambuff* for each packet, which consists of a unique pointer to an array of reference-counted buffers. Whenever a packet must be shared, a new streambuff is created, whose array points to the same databuffers as the original (after increasing their reference counts). This pproach allows for quickly appending and prepending data to a packet, and requires copying packet buffers only when they are both shared and mutated. Linux `skbuff`s have similar properties, but lack the safety benefits ensured by our type system. Once again, programming with our constructs was not overly onerous. Out of 13000 lines, we added 76 annotations, used swap 46 times, and `alias` 66 times, of which 71% were automatically inferred. We are exploring further improvements to reduce this burden.

Figure 2 illustrates the memory usage of MediaNet when forwarding 5000 4 KB packets over Gigabit Ethernet. The *gc* configuration uses garbage collection for all packet data, whereas *gc+free* is as described above. This graph has the same format as the graph in Figure 1, but also shows the heap and reference-counted regions. Reserved memory for the *gc+free* case is not shown.

The *gc* configuration exhibits a sawtooth pattern, where each peak roughly coincides with a garbage collection. The *gc+free* configuration uses and reserves far less memory (128 KB as opposed to 840 KB for reserved memory, and 8 KB as opposed to 420 KB of peak used memory). There is some initial heap-allocated data that remains throughout the run, and the reference-counted and unique data (the small line at the bottom) never consume more than a single packet's worth of space, since each packet is freed before the next packet is read in.

4. CONCLUSIONS AND FUTURE WORK

Cyclone supports a rich set of safe memory-management idioms for users unwilling to use only automatic techniques. Together, these idioms represent significant progress toward the goal of enforcing sound, manual memory management. Preliminary experiments show the constructs can facilitate good performance, without imposing an undue burden.

We are currently focusing on improving the ease of use of our techniques. Firstly, we are designing tools to enable semi-automatic porting of C code to Cyclone that can use unique pointers. We are considering traditional, whole-program static analysis, augmented by dynamic information. Second, we are improving intraprocedural inference in Cyclone. For example, a more general constraint-based inference should be able to discover many of the `alias` statements not already inferred. Similarly, support for a `restrict` mechanism in the style of Aiken et al. [1] should reduce the need for swapping, at the cost of more annotations. Finally, the aliasing restrictions on unique and reference-counted pointers can make error messages hard to understand. We are exploring visualization tools and ranking heuristics to help programmers find the likely cause of an error.

5. REFERENCES

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, 2003.

[2] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[3] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.

[4] Michael Hicks, Adithya Nagajaran, and Robbert van Renesse. MediaNet: User-defined adaptive scheduling for streaming data. In *IEEE OPENARCH*, 2003.

[5] David Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, 2001.

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