Real-Time Garbage Collection in a Multimedia Programming Language

Position Paper
OOPSLA 1993 workshop on Memory Management and Garbage Collection
September 19, 1993

Wade Hennessey
Kaleida Labs
1945 Charleston Road
Mountain View, California 94043
Email: wade@kaleida.com, Phone: 415-966-0425

1.0 Overview

In order to facilitate the development of platform independent Multimedia applications, Kaleida Labs is developing a programming language called ScriptX[ScriptX]. ScriptX is a dynamic, object-oriented language which includes features such as multiple threads of execution and real-time garbage collection. This paper describes the implementation of the memory management portion of the system.

ScriptX is actually implemented in Objects In C (OIC). OIC is essentially a library which enables object-oriented programming in C. OIC is loosely related to Smalltalk and CLOS, and includes multiple-inheritance and multi-argument method dispatch in generic functions. We are writing a real-time garbage collector for OIC, which in turn provides ScriptX with garbage collection because ScriptX objects are simply OIC objects.

2.0 Design Constraints

The OIC garbage collector was designed with the following constraints in mind:

- Real-time performance - Because multimedia applications must manipulate audio and video data in real-time, it is essential that the garbage collector work is smoothly interleaved with mutator work in increments that are guaranteed to never exceed a few milliseconds.
- No compiler support - In order to enhance portability, OIC is based on ANSI C, thus allowing it to run on any system with an ANSI C compiler. However, we cannot modify the C compiler or even provide our own preprocessor to be run over OIC code.
- Minimal hardware - Although ScriptX is designed to run on hardware ranging from small consumer “settop” computers all the way up to workstations, we must design to the lowest hardware level. Because settop boxes are intended to be inexpensive con-

sumer electronics devices which connect to a cable service and a television, we cannot expect any virtual memory support and may have to run with only 1 or 2 megabytes of RAM. ScriptX itself, including the garbage collector, may be supplied in ROM.

3.0 Implementation

We have chosen to use a variation of Bakers “Treadmill” algorithm [Baker]. Our collector uses a write-barrier to synchronize the mutator and the garbage collector, and was inspired by a similar variation of the Treadmill described in [Wilson]. Because we are providing garbage collection for both OIC objects as well as regular heap allocated C data such as structures and arrays, we must deal with several problems:

- Conservative Collection - While OIC provides information about the location and type of pointers in class instances, it does not identify pointers in objects such as structures, global data, or thread stacks. Thus, we must conservatively scan objects which may contain unidentified pointers. In order to find these pointers, we look at all possible pointers in an object from start to end in increments equal to the pointer alignment of the processor we are using. For example, on a SPARC based machine, it is possible to only look for pointers aligned on 4 bytes boundaries, while on a 68K based Mac we must look for pointers aligned on 2 byte boundaries.
- Interior Pointers - It is also quite common in OIC to have a pointer to the middle of an object such as a string or a structure. Such pointers are commonly called “Interior Pointers” and must be easily identified and converted into a pointer to the header of the enclosing object. Thus, even if the only remaining pointer to an object is an interior pointer to an object, we are able to correctly identify and mark the entire object.
- Write Barrier - Because we have no compiler or preprocessor support, we must rely on a programmer assisted write barrier. We currently require the programmer to use a macro to store pointers into global data and heap allocated objects. However the write barrier is not needed for pointer writes into a thread stack or the registers.

A garbage collector is normally invisible to programmers. However, in addition to the write barrier, the programmer can “assist” the garbage collector in a few other ways:

- Finalization - Some objects cannot be freed without first performing some extra work. For example, a file stream may need to release an underlying operating system resource such as a file descriptor before being freed. Similarly, an object may contain pointers to storage which is manually allocated within the operating system, and this storage must be manually deallocated before freeing the object. In order to handle these cases, objects may be registered for “Finalization”, allowing programmer provided OIC code to perform these final actions before the object is actually freed by the garbage collector.

  Manual Deallocation - We allow programs to bypass the garbage collector and manually deallocate objects.

4.0 Status

Our initial implementation of the collector runs on three platforms: SPARC based Sun Workstations, the Apple Macintosh, and the IBM PC. However, the Macintosh is our primary development platform, and thus our current work is geared towards optimizing performance on he the Mac.

The collector is extremely conservative, and allocates all new objects black and uses a "snapshot" write barrier to intercept all writes into the OIC heap and into the global data section. On a Quadra 800 Macintosh, the current system has an average pause time of 700 microseconds. Although the current collector is usable, there are many areas we would like to improve:

- Maximum Pause time - Because the thread stacks are not subject to the write barrier, we must scan each stack atomically. Unfortunately, this means that the maximum pause time of the collector is dependent upon the size of the thread stack with the largest number of pointers. In practice, this leads to a maximum pause time of 8 to 30 milliseconds. However, there is no way to *guarantee* a maximum pause time.
- Memory fragmentation - We currently allocate all memory in sizes which are powers of two, with the first 8 bytes of storage being used by the collect to store link pointers and other information. Current measurements indicates that only 60% of the memory allocated is actually used directly by OIC code. The remaining memory is either wasted or attributed to GC overhead.
- Write barrier overhead - The write barrier appears to be the dominant cost of our real-time collector. Compared to a non-real time version of the collector that we have also implemented, the real-time collector requires 50% to 100% more time, almost all of which is spent in the write-barrier. Additionally, the write barrier is difficult to implement in a system like ours without compiler support, although we currently have a "lint" like program to flag violations of the write barrier. These include not only assignment statements, but also less obvious culprits such as aggregate structure assignments and calls to library routines such as `memcpy`.
- Freeing memory - Because we have a conservative collector, we must currently change the color of all memory about to be freed from white to a "free" color. If we did not do this, then at any given time some of the objects in the free list would appear to be white, and thus random data which the conservative collector treats as pointers could cause the collector to start stealing objects from the free list! Thus, the time it takes us to "flip" the collector is proportional to the number of objects which are being freed.

5.0 Future Work

Given the status of the current collector, we plan to focus our future efforts on a few key areas:

- Write Barrier - In order to reduce write barrier overhead, we do not require initializing writes to go through the barrier. However, we may also start to require pointer writes to the stack to also go through the barrier. This will allow us considerably more freedom

with our allocation color and write barrier strategy. It's possible that white allocation of new objects coupled with an incremental-update write barrier will be considerably more efficient than our current scheme.

- Compaction - Because multimedia applications tend to allocate some very large objects to represent audio and video information, we plan to investigate strategies for real-time compaction of storage as well as allocating large objects in finer grains than the current power of two scheme.
- Generations - One of the most glaring inefficiencies in the current collector is the amount of work it does repeatedly collecting data which will never become garbage. Adding generations to the collector is an obvious way to reduce this work.
- Root registration - The collector also spends too much time conservatively scanning the global data section of ScriptX. On the Mac, the global data section currently occupies 100 kbytes which must be incrementally scanned two bytes at a time looking for possible pointers. It would be much more efficient to either implicitly or explicitly register global data which may contain pointers, thus reducing part of the constant overhead inherent in each collection.

REFERENCES

[ScriptX] Kaleida Labs. *ScriptX Architectural Overview.*

[Wilson] Paul R. Wilson, Mark Johnston. *Truly Real-Time Non-Moving Implicit-Reclamation Garbage Collection (ROUGH DRAFT -- comments welcome)*. Preliminary report available on **cs.utexas.edu** via anonymous FTP. Dated March, 8 1993.

[Baker] Henry G. Baker Jr. *The Treadmill: Real-time Garbage Collection without Motion Sickness*. SIGPLAN Notices 27(3):66-70, March 1992.