Realtime Garbage Collection in JamaicaVM

This article gives an introduction into JamaicaVM realtime Java technology.

The modern object oriented programming language Java relies on the concept of automatic garbage collection (GC) to provide safe memory management. Garbage collection protects the user from hard to debug programming errors and difficult to maintain programs on the one hand and provides secure execution of untrusted code on the other hand. C-style explicit memory allocation and reclamation techniques are a major obstacle for object oriented programming.

Most current garbage collector implementations still fail to give useful or proven realtime guarantees on their runtime performance. While Java is applied to a growing extent for the development of systems that require realtime guarantees, classical GC is not applicable in these systems. The Real-Time Specification for Java avoids automatic memory management for hard realtime tasks.

The JamaicaVM implements this specification and the scoped memory management that it provides as an alternative to GC, but JamaicaVM also provides realtime garbage collection (RTGC) as a simpler alternative to the often complex assignment rules that are required by the use of non heap memory. Only the combination of RTGC and the RTSJ provide the full ease of development of Java for the development of realtime code.

Garbage Collection Basics

The most widely used algorithms for GC in Java are based on the mark and sweep or mark sweep compact algorithms. GC with these algorithms is a cyclic process. Normally, a single GC cycle has four phases: root scanning, mark, sweep, and compaction.

Root Scanning
All objects on the heap referenced by root variables are marked. Root variables are all variables that are not stored in the heap itself, i.e., variables on the runtime stacks of all threads, within processor registers, or in global variables that are stored outside of the garbage collected heap.

Mark
During the mark phase, all objects on the heap that are reachable from marked objects will be marked as well. This phase continues until no new objects can be marked, i.e., until all objects that are reachable from root references are marked.

Figure 1: Standard GC phases
Sweep
After the mark phase, all allocated objects that have not been marked are known to be unreachable. These objects are garbage; their memory can be reused. Thus, during the sweep phase, the heap is traversed and all unmarked objects are added to free memory.

Compact
After the sweep phase, free memory will in general be noncontiguous. This fragmentation of memory makes small ranges of free memory unavailable for allocations of larger objects and leads to reduced memory use. The common solution to this problem is the use of an additional compaction phase that moves allocated objects such that free memory forms a single contiguous range.

In an incremental garbage collector, these four phases must be executed in small increments of garbage collection work while the application continues its execution and modifies the heap. For the garbage collector to be hard realtime, the amount of work to be done in these increments must be predictable and the system must guarantee that the garbage collector recycles sufficient memory to satisfy all application allocation requests.

Realtime Garbage Collection
The realtime garbage collector employed by JamaicaVM enables the use of automatic memory management for realtime applications with tight timing constraints. This extraordinary facility is accomplished through a special memory layout and careful implementation. The work of garbage collection is also shared among all application threads so as to minimize its impact on the overall system.

Avoid Root Scanning Phase
During root scanning, all objects that are referenced by local variables or processor registers have to be marked. Marking typically requires suspending the examined thread, which leads to delays that are hard to predict. The solution in the JamaicaVM is, instead of having a dedicated root scanning phase, the runtime system ensures that copies of all root references are stored on the heap whenever a the garbage collector may become active.

The compiler automatically generates code to store the root references and to remove the stored references of roots that are deleted. The data structures that are used to store the copied references are all reachable from a single root object. Consequently, all stored root references are reachable from this single root object and the root scanning phase is reduced to marking this single root object. Marking the copied root references becomes part of the garbage collector’s mark phase. To avoid inconsistencies when storing roots while marking is going on, write barrier code ensures correct marking of stored roots.

Specific optimizations have been developed for JamaicaVM, such that the runtime cost is minimal.

Non-Fragmenting Object Model
Allocation and deallocation of objects of different sizes in a system with dynamic memory management causes fragmentation. Fragmented memory is unused memory in small, non-contiguous ranges that cannot be used to satisfy larger allocation requests. Fragmentation can cause severe loss of memory utilization that is hard to predict. Current implementations often use compaction to fight fragmentation. Memory compaction can remove fragmentation, but it destroys realtime guarantees due to the need to
move arbitrarily large objects. Since moving objects seems to inherently contradict the predictability requirements of realtime systems, the compaction approach has been dropped in JamaicaVM. Instead, an object model based on non-movable blocks of a fixed size is used. Objects and arrays are constructed from several blocks that need not be contiguous in memory. Figure 2 illustrates the structure of a Java object that is spread over 3 blocks of equal size.

The surprising result is that this object model permits an efficient runtime performance: since Java objects tend to be small, most object accesses turn out to refer to the first block of an object. Obtaining the value of these references can be performed in a single memory access. Since the object layout is known at compile time, the time required for any object access can be determined statically, enabling worst-case execution time analysis.

With typical large Java applications, one sees an average of about 1.1 memory accesses per object access. In contrast, a compacting garbage collector has to use an additional indirection, so called handles, to be able to move objects around. This additional indirection results in 2 memory accesses for every object access.

### Incremental Mark-and-Sweep

With the root scanning phase eliminated by copying of root references to the heap and the compaction phase removed by a nonfragmenting object model based on fixed size blocks, the remaining tasks for the garbage collector from Figure 1 are the Mark and the Sweep phases shown in Figure 3. Now, these phases can be performed in incremental steps: one incremental mark step consists of marking of one single block, and one incremental sweep step consists of sweeping one single block. These steps are uniform and very small. The worst-case on a PowerPC processor is only about 150 machine instructions.

### When to Run the GC

The decision of when to run the garbage collector in a realtime system has to ensure that the runtime spent for garbage collection is predictable and limited, while also guaranteeing that sufficient memory is recycled such that all allocation requests can be satisfied. Current implementations typically use a separate thread for the garbage collector. This approach, however, makes predictions on the overhead and efficiency of the garbage collector, as required in a realtime system, extremely hard.

JamaicaVM does not use a separate thread for garbage collection. Instead, the garbage collector is activated within an application thread whenever this thread allocates memory. Performing the garbage collection work at each allocation automatically runs the garbage collector more aggressively whenever more memory is allocated by the application. There is no garbage collection overhead during times when no allocation is performed and high-priority threads which do not perform allocation will never be delayed due to garbage collection. JamaicaVM provides tools to determine the worst-case execution time of the garbage collection work required at an allocation. The implementation ensures that the garbage collection work is sufficient to reclaim memory fast enough for the application not to run out of memory.

### Summary

JamaicaVM’s realtime garbage collector runs in very small, incremental steps. These steps are performed at a time which is predictable by the user: whenever an object is allocated. Determining the worst-case execution time for object allocation is also possible. These features combined permit the use of garbage collection even in realtime systems. Realtime code that is executed in high priority threads is completely unaffected by garbage collection work, and it is even possible to perform dynamic memory allocation in realtime code since a worst-case execution time for an allocation can be determined.