Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor

Fridtjof Siebert
siebert@real-time-systems.de

Abstract

We show the difficulties that arise for the implementation of a real-time garbage collector (GC) in a multi-threaded system. A mechanism for synchronization between threads is proposed for a single processor system. It is shown how this mechanism can be used to maintain exact information on roots, to do incremental or even constant-time root-scanning and to allow pre-emption of GC activity.

1. Introduction

The difficulties caused by synchronization between the GC and several mutator threads have so far made an implementation of a hard real-time GC for such widely used languages as Java impossible. By real-time GC, we mean a system that allows worst case execution times to be derived for any task that is performed, especially operations like memory accesses, reference assignments, modifications of the root set and allocations. For such a system to be useful, the worst case performance must be very small. We limit our analysis to the problems that arise by the co-ordination between threads.

For incremental scanning of the memory-graph to be correct, most GC algorithms rely on a write-barrier to inform the collector about changes made by the mutator [1]. Write-barrier code typically must not be pre-empted by GC activity. On standard hardware, it cannot be implemented using a single atomic instruction. Additional locking code is required to ensure atomicity.

There exist very few GCs that do not require this locking. One is presented in [2], but this algorithm is limited to a single mutator thread. Another one is [3]: This approach uses a significantly more complex write-barrier, and it uses a complex synchronization of all threads at the beginning of each GC cycle. This approach is nevertheless similar to the mechanism proposed here as it requires insertion of *synchronization points*.

Another difficulty that arises in multi-threaded systems is scanning values on the stacks for root pointers. Current implementations often use conservative scanning techniques [4] that do not require exact information on the location of references. This conservatism makes such implementations unusable for security relevant domains and for applications that require guarantees on the performance of the GC to be able to satisfy their allocation requests or to meet real-time deadlines.

We assume an incremental implementation of a mark-and-sweep GC [5], but the proposed mechanism is just as well applicable to copying collectors [6]. When we talk about *native threads*, we mean threads that can be scheduled at any time, like it is usually the case for kernel-level threads.

2. Synchronization points

The described difficulties are complicated by the use of native threads that impose no restrictions on when thread switches may occur. If GC work is required, there is very little information available on the state of the threads.

The solution we propose here is to prohibit thread switches at arbitrary points during execution for mutator threads. Instead, scheduling should occur only at *synchronization points* that are automatically inserted in the code by a compiler or virtual machine. At such a *synchronization point*, a test of a global variable indicates if a thread switch is required, and some code is executed if this is the case. Here is the required code for a *synchronization point* as as pseudo-C code:

```
if (synchronization_required == true) {
  ... record thread's state ...
  V(global_thread_semaphore); /* thread switch */
  P(global_thread_semaphore);
}
```

The thread scheduler has to set the flag *synchronization_ required* whenever a different thread than the currently running one should become active.

These *synchronization points* have to be inserted frequently enough so that pre-emption of threads is possible with a minimal delay. This is required in real-time applications where high priority tasks have to be able to pre-empt lower priority tasks within a fixed delay. A compiler or virtual machine implementation can generate code in a way that guarantees frequent execution of the *synchronization points*. This can be achieved by generating this code within all loops, at (potentially recursive) calls and within long sequences of linear code. For a given platform it is possible to guarantee an upper bound for the length of the time interval between two *synchronization points*.

The *synchronization points* can be realized using native threads and a global semaphore [7] that has to be acquired by a mutator thread to execute. To allow a thread switch at a *synchronization point*, this semaphore will be released and directly reacquired to allow a different thread to take over the processor, as shown above.

When *synchronization points* are used, the restrictions imposed on the mutator by the GC are relaxed significantly for the code between two *synchronization points*. There are three main aspects:

1. The invariants required by the GC do not have to hold in between two *synchronization points*. A typical GC invariant is*There are no pointers from a black object to a white object.* It is sufficient to restore the invariant by the time the next *synchronization point* is reached. This gives freedom to optimizing compilers allowing them to modify all the code in between *synchronization points*.

2. No locks are required to modify the memory graph or global data used for memory management. Especially write-barrier code and allocation of objects can be performed without the need for locks on the accessed data structures since all other threads are halted at *synchronization points* and are guaranteed not to modify this data at the same time.

3. No exact reference information is required, since root scanning cannot take place in between *synchronization points*.

3. Garbage Collector

To run the GC, we have a choice between two possibilities: Either, we have the GC running as a separate thread parallel to the mutators, or we intertwine GC and mutator activity doing GC increments within the mutator threads. Since the amount of GC work that is required can be expressed as a function of the amount of allocation that is going on [8], the latter solution seems to be preferable. This also allows a fair accounting for GC work, when increments of the collection work are performed at allocations and the thread performing an allocation has to provide the CPU-time for the GC activity required to satisfy the allocation request. Furthermore, the GC code itself can make use of *synchronization points*, so that a higher priority thread can become active even while the GC is scanning memory. A simple approach would be to add *synchronization points* after scanning of each grey object.

4. Root scanning

One of the biggest problems a real-time GC has to deal with is scanning of root pointers (reference values that are stored locally on the stack or in registers of a mutator thread). Whenever the GC needs to scan the root pointers, exact information on their whereabouts is required. Fortunately, this can happen only if synchronization is actually required. Since the set of life variables is statically determinable, the recording of this set can be done by a single write to memory, that is done conditionally at a *synchronization point* only if synchronization is actually required.

Call points

Unfortunately, it is not sufficient to have root information on the routine that is currently being executed, but the same information is also required for all methods that are currently active, so that the list of active stack frames can be traversed and scanned. This requires the mutator threads to store information on the caller's root pointers on calls.

Incremental root scanning

To scan a stack for roots, the thread's execution must be stopped. This can introduces a significant worst case delay for all mutator threads. So it might be preferable for the GC to proceed incrementally, e.g., on a per-stack-frame basis.

Mutator-threads can generate and delete stack-frames while they are running, and they are allowed to modify the references of already scanned frames, making rescanning of these frames necessary. This can be done by adding a flag to every frame that indicates the need for rescanning.

Constant time root scanning

Even though incremental root-scanning can limit the worst case delays during the root-scanning phase of the GC cycle, there remain several difficulties: The GC implementation is complicated by this mechanism, large stack-frames can cause long worst case delays and it is hard to guarantee progress since a thread might continuously force rescanning of its stack-frames.

To solve these problems, we want to take the task of scanning the stack-frames completely away from the GC. Instead, the mutator threads should guarantee that at a *synchronization point*, any reference that is used locally by a thread is also present on the garbage collected heap. This can be ensured by storing all life reference variables of the current routine into the heap on *synchronization points* and on calls. Per-thread preallocated memory sub-pools or stacks can be used for this.

When the references are stored, GC constraints like write-barriers have to be obeyed. An implementation of this mechanism has to ensure that storing of these references is efficient, not to increase the overhead for calls unreasonably. We have run several large Java applications (a compiler, SwingSet and HotJava) to collect information on the required additional code on calls. The applications had in average between 2.02 and 2.73 life references on a call, adding the overhead of storing 2 to 3 references, which appears to be reasonable.

Having all reference values present on the heap, it is sufficient to have a single root pointer in the system that is scanned at the beginning of the GC cycle. It just has to be ensured that all memory used for local reference variables is reachable from this single root pointer. The root scanning time is constant and very small.

5. Conclusion

We have presented a new way to synchronize mutator threads and GC on single processor systems. We have shown that this allows to provide information for exact incremental GC that is hard to achieve when native threads are used directly. Since we have limited our analysis on the synchronization aspects, for a complete real-time GC implementation many other decisions have to be made. A suitable incremental algorithm has to be chosen, among the classical ones are [5], [6] and [9]. The algorithm has to be implemented carefully to be exact and to have a good worst case performance behaviour. Then, a mechanism that guarantees sufficient GC progress has to be used, like the ones presented in [10] and [8]. Another problem that needs to be addressed is fragmentation.

6. References

[1] Pekka P. Pirinen: *Barrier techniques for Incremental Tracing*, ISMM, 1998

[2] Lorenz Huelsbergen, Phil Winterbottom: *Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization*, ISMM, 1998

[3] Damien Doligez, Georges Gonthier: *Portable, Unobtrusive Garbage Collection for Multiprocessor Systems*, POPL, 1994

[4] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988

[5] Edsgar W. Dijkstra, L. Lamport, A. Marin, C. Scholten and E. Steffens: *On-the-fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM, 21,11 (November 1978), p. 966-975

[6] Henry G. Baker: *List processing in Real Time on a Serial Computer.* Communications of the ACM 21,4 (April 1978), p. 280-294,

[7] Edsgar W. Dijkstra: *Cooperating Sequential Processes*, Technical Report EWD-123, Technical University Eindhoven, 1965

[8] Fridtjof Siebert: *Guaranteeing Non-Disruptiveness and Real-Time Deadlines in an Incremental Garbage Collector*, ISMM, 1998

[9] Henry G. Baker: *The Treadmill: Real-Time Garbage Collection without Motion Sickness*, ACM SIGPLAN Notices 27,3, 1992, pp. 66-70

[11] Paul R. Wilson: *Uniprocessor Garbage Collection Techniques*, 1998