Fridtjof Siebert
siebert@real-time-systems.de

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

Abstract

We show the difficulties that arise for the implementation of a real-time garbage collector in a multi-threaded system. A mechanism for synchronization between threads and the garbage collector activities is proposed for a single processor system. It is shown how this mechanism can be used to maintain exact information on roots, to implement efficient write-barriers, to do incremental or even constant-time root-scanning and to guarantee short pre-emption time of garbage collector activity. Special aspects of an implementation for Java that are affected by this mechanism will also be addressed. Finally, experimental data is presented to show that the proposed mechanisms can efficiently be used in real programs.

Introduction

The difficulties caused by synchronization between the garbage collector (GC) and several mutator threads have so far made an implementation of an exact, incremental, hard real-time garbage collector for such widely used languages as Java impossible. Because of the lack of reliable GC, Java cannot be used in a wide area of real-time applications.

When we are talking about real-time garbage collection, we mean a system that allows worst case execution times to be derived for any task that is performed. In the context of GC this means that the worst case performance of operations like memory accesses, reference assignments, modifications of the root set and allocations are known. For such a system to be useful, this worst case performance must be small, else the worst case performance of a bigger task consisting of several of such operations quickly becomes unacceptable. This definition of real-time GC is very similar to that presented in [Nilson94]. The difference of our solution is that we do not try to achieve real-time GC by the application of specialized hardware, but by means of software alone. Additionally, we do not want to present a complete GC implementation here, but instead limit our analysis to the problems that arise by the presence of threads and the required co-ordination between GC and mutator threads.

Multi-threaded systems complicate garbage collection significantly, but threads are also a required prerequisite to do serious development in languages like Java [AG98]. Life is further complicated when incremental and exact garbage collection [Wilson98] is required: The garbage collector activity is intertwined with the execution of the main program (the mutator) and communication between GC and the mutator-threads is required.

For incremental scanning of the memory-graph to be correct, most garbage collection algorithms rely on a write-barrier to inform the collector about changes made by the mutator [Pirinen98]. The write-barrier code typically requires writing a word or a bit conditionally to indicate the change of the graph, in addition to the actual memory write required for an assignment.

Write-barrier code usually 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 garbage collectors that do not require this locking. One is presented in [HW98], but this algorithm is limited to a single mutator thread and not applicable to multiple mutator threads. Another one is [DG94]: This approach uses a significantly more complex write-barrier that marks both references, the assigned and the deleted one, and it uses a complex synchronization of all threads at the beginning of each garbage collection cycle. This approach is nevertheless similar to the mechanism proposed here as it requires insertion of *synchronization points*, but with the difference that scheduling is not restricted to these 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 [Barlett88] that do not require exact information on the location of references. This conservatism makes it impossible for the implementation to give guarantees on the amount of memory that will be recycled and the effectiveness of defragmentation. Even worse, the system becomes vulnerable to denial-of-service attacks that exploit knowledge about the conservative implementation of the garbage collector [BLT98]. These deficits make such an implementation unusable for security relevant domains and for applications that require guarantees on the performance of the garbage collector to be able to satisfy their allocation requests or to meet real-time deadlines.

To achieve small pause times in an incremental garbage collector it might be required to incrementally scan the stacks of the involved threads for root pointers. This requires not only exact information on the references in the stacks of all threads, but also information on the changes made to the stack-frames between incremental root scanning steps.

In this paper, we assume an incremental implementation of a mark-and-sweep garbage collector [Dijkstra78], but the proposed mechanisms should just as well be applicable to copying collectors [Baker78].

When we talk about *native threads*, we mean threads where scheduling can occur at any time, like it is usually the case for kernel-level threads.

Synchronization points

All the difficulties described above are complicated by the use of native threads that impose no restrictions on when thread switches may occur. If garbage collection work is required, there is very little information available on the state of the threads. It might be possible to obtain more accurate information by determining the program counter of a thread that is not running and using it to obtain information on the thread's state. But this will be rather complicated and very platform dependent.

The solution we want to propose here is to prohibit thread switches at arbitrary points during execution for mutator threads. Instead, a scheduling should occur only at certain *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. As much code as possible should be executed conditionally, in a way that there is no additional overhead as long as synchronization is not required. **Figure 1** shows the required code for a *synchronization point* as pseudo-C code. The thread scheduler has to set the flag *synchronization_required* whenever a different thread than the currently running one should become active.

```
if (synchronization_required == true) {
   ... store information on thread's state ...
   ... allow thread switch ...
}
```

Figure 1: Basic *synchronization point* code.

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 like [TurboJ] or [JDK] 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*.

This is a software approach to a similar mechanism implemented in hardware for the scheduling of processes on transputer processors [Inmos93]. In a transputer, certain instructions like jumps are so-called *descheduling points* and scheduling of other processes can only take place at these instructions.

In software, the *synchronization points* can be realized using native threads and a global semaphore [Dijkstra65] 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 mutator thread or the GC to take over the processor, as in

```
V(global_thread_semaphore);
P(global_thread_semaphore);
```

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

1. Invariants do not need to hold

The invariants required by the garbage collector do not have to hold in between two *synchronization points*. A typical GC invariant [Pirinen98] 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. Compared to more fine-grain synchronization techniques, this gives freedom to optimizing compilers allowing them to modify all the code in between *synchronization points*. Even code modifying the heap, like the marking within a write barrier or allocation of an object, can profit from optimizations like redundancy elimination [KR94], instruction scheduling [HKHW96], etc. The invariant can be destroyed temporarily by these optimization in a way that would otherwise break the system's consistency.

2. No locks required

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 threads are halted at *synchronization points* and are guaranteed not to modify this data at the same time.

3. No exact reference information required

Even for exact garbage collection, information on reference values is not required, since root scanning cannot take place in between *synchronization points*. Any local variable or register that is used to hold a reference and that has a life-span that lies completely within two *synchronization points* does not need to be added to the root set.

Garbage Collector

To run the garbage collector, we have a choice between two possibilities: Either, we have the garbage collector running as a separate thread parallel to the mutators, or we intertwine garbage collector and mutator activity doing garbage collector increments within the mutator threads. Since the amount of garbage collection work that is required can be expressed as a function of the amount of allocation that is going on [Siebert98], the latter solution seems to be preferable. This also allows a fair accounting for garbage collection 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 garbage collector activity required to satisfy the allocation request.

The garbage collector code itself can make use of *synchronization points*, so that thread switches can be allowed even while the garbage collector is scanning memory. A simple approach would be to add *synchronization points* after scanning of each grey object. If a low priority thread has to perform a garbage collection increment that consists of scanning several objects, high priority threads can still preempt this process after scanning of each of these objects. If the time required to scan one object is bounded, the pre-emption time will also be bounded.

Root scanning

One of the biggest problems a real-time garbage collector has to deal with is scanning of root pointers. Root pointers are reference values that are stored locally on the stack or in registers of a mutator thread. Whenever the garbage collector might want to scan the root pointers, exact information on their whereabouts is required. Fortunately, this can happen only if synchronization is actually required. **Figure 2** illustrates how this information can be provided conditionally at a *synchronization point*. Since the set of life variables is statically determinable, the recording of this set can be performed by a single write to memory. It is executed only in the case that synchronization is actually required, so the average run-time cost is minimal.

```
if (synchronization_required == true) {
   current_stack_frame.roots = set of life locals/registers
                               that hold references;
   ...
}
```

Figure 2: Recording the set of life variables as roots for the garbage collector at a *synchronization point*.

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.

Figure 3 illustrates the code that would be required to perform a call, namely registering the root set of the caller and performing the actual call. The garbage collector has to be able to traverse the list of stack-frames during root scanning. If the single frames are not linked anyway, the size of a stack frame can be stored with the root information. This will allow finding the address of the following frame without the need of executing additional code at run-time.

```
current_stack_frame.roots = set of life locals/registers
                            that hold references;

Result = CallMethod(arguments)
```

Figure 3: Code for root scanning and synchronization at a call point.

Incremental root scanning

The information available so far allows the garbage collector to find reference used locally within the stack of each thread. But to scan a stack for roots, the thread's execution must be stopped. Since the time required for scanning of a stack can be fairly long (in $O(n)$ for a stack of size n), this introduces a significant worst case delay for all mutator threads that won't be tolerable for many real-time applications. It would be preferable if the garbage collector could proceed incrementally, e.g., on a per-stack-frame basis, so that mutator-threads can become active after only a part of their roots have been scanned.

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. Fortunately, in most modern languages like Java [AG98] or Eiffel [Meyer92], it is not possible to modify the contents of stack frames other than the current stack frame. Rescanning is only needed for the frames of those methods that were active since they have last been scanned.

A simple solution is to add a flag that indicates the need for rescanning. It is again sufficient to provide exact information on the need for rescanning only at *synchronization points* and at calls. This can be done by setting a boolean field in the current stack-frame, as in

```
current_stack_frame.rescan_required = true;
```

This code has to be added to the conditional part of the *synchronization point* and to every call. The garbage collector will reset this value to *false* whenever it has scanned a stack-frame. The additional assignment can actually be removed by a slightly different interpretation of *stack_frame.roots*[1], so that there is no additional overhead within the mutators.

[1] If the garbage collector sets the root set pointer of all scanned frames to some special value like *null*, this can be used to tell scanned frames from those that still need to be scanned or that need to be rescanned. Since the root set pointer is assigned a non-*null* value by the mutator whenever *rescan_required* is set to *true*, the expressions '*stack_frame.rescan_required*' and '*stack_frame.roots != null*' will be equivalent.

Constant time root scanning

Even though incremental root-scanning can limit the worst case delays during the root-scanning phase of the garbage collection cycle, there remain several difficulties.

1. The garbage collector implementation is complicated by this mechanism.
2. Large stack-frames can still cause long worst case delays.
3. It becomes hard to guarantee garbage collection 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 garbage collector. 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 task. **Figure 4** illustrates code that is required to do this at a *synchronization point*. Similar code is needed at a call.

```
if (synchronization_required == true) {
   local_reference_stack.push(local_ref1);
   local_reference_stack.push(local_ref2);
   local_reference_stack.push(local_ref3);

   ... allow thread switch ...

   local_reference_stack.pop(3);
}
```

Figure 4: Storing three locally used references on the heap at a *synchronization point*.

When the references are stored, garbage collector 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. As we will see below, there are usually only few life reference variables to be stored, so that the run-time cost is low.

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 garbage collection 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.

Example

As noted above, the use of *synchronization points* as described allows for several optimizations that are not possible when native threads are used directly in a garbage collected system. **Figure 5** illustrates a small Java method to insert an element at the head of a list.

```
class Node {
  Node next, prev;
}
class List {
  Node head;
  void insertHead(Node node) {
    node.next = head;
    head.prev = node;
    head      = node;
  }
  ...
}
```

Figure 5: Java method *List.insertHead*.

We assume that a write-barrier that marks the newly assigned reference grey is used. **Figure 6** shows pseudo-C code that is needed to implement this method using a traditional incremental collector and native threads.

```
void inserthead(List this, Node node) {
   atomic { node.next = this.head; mark_grey(this.head); }
   atomic { head.prev = node;      mark_grey(node);      }
   atomic { head      = node;      mark_grey(node);      }
}
```

Figure 6: *List.insertHead* using native threads.

Using *synchronization points* and the optimizations that become possible by their usage, the code will look like the one presented in **Figure 7**. Here, no atomic instructions are required, the code is automatically atomic since it is not interrupted by *synchronization points*. Because it is a short leaf method, no code for a *synchronization point* is required within the method. And finally, the garbage collector marking can profit from compiler optimizations (like common subexpression removal) that can easily detect that the second greying of *node* is not necessary.

```
void inserthead(List this, Node node) {
   node.next = this.head; mark_grey(this.head);
   head.prev = node;      mark_grey(node);
   this.head = node;      /* node already marked grey */
}
```

Figure 7: *List.insertHead* using *synchronization points*.

Optimizing the synchronization overhead

Several simple optimizations can be used to reduce the run-time and code-size overhead caused by the introduction of *synchronization points*. Some of them are presented here:

Loops that have a known number of iterations (like simple *for*-loops) do not necessarily need the *synchronization point* code. The compiler can determine the amount of time that the execution of the loop will require. In case the loop finishes quickly enough, no *synchronization point* is needed within its body, the loop can instead be treated as if it was a linear stretch of code.

The same holds for calls to leaf-methods: If the call target is known to the compiler and the execution time is known as well, the compiler might decide that synchronization is not needed for a call. This is very likely to hold for a call to the method presented in **Figure 5**. This optimization can be extended for non-recursive methods (those that only call leaf methods or other non-recursive methods): There is no need for a *synchronization point* if the compiler can determine the time needed to execute the call and the time is small enough.

Small loops that need a *synchronization point* within their body might suffer more significantly from the additional code that has to be executed on every iteration. A compiler can easily reduce the overhead in this case by unrolling the loop a few times. The unrolled loop still requires only a single *synchronization point*, reducing the overhead to 50% by unrolling once, to 33% by unrolling twice, etc.

A compiler that has full control over the machine code that is being generated might assign a register to the flag *synchronization_required*. The thread scheduler will have to indicate the need for synchronization by directly setting this register. This approach will avoid the need for a memory access and significantly reduce the overhead by *synchronization points*.

A trade-off can be found between the overhead introduced by *synchronization points* and the speed of pre-emption required by the application. Allowing the compiler to execute longer stretches of code without having a *synchronization point* can avoid the overhead due to synchronization, while requiring a check for synchronization very frequently, like every *100* machine cycles, will allow very quick pre-emption.

Implications for a Java implementation

We present three implications that arise when *synchronization points* are applied to a Java [LY96] implementation that do not have a direct relation to garbage collection: The efficient implementation of Java monitors, difficulties that arise by the presence of native code and the object layout.

Java monitor implementation

Due to the frequent use of monitors, their implementation has an important impact on the overall performance of a Java implementation. In [BKMS98] it has been shown that an efficient monitor implementation can speed up typical Java applications by an average factor of 1.22.

In a system that limits thread switches to *synchronization points*, the implementation of a Java monitor is simplified significantly. Since the code between *synchronization points* is automatically atomic, no locking mechanism is required to request or change a monitor's state. Special hardware support as provided by atomic *test-and-set*, *compare-and-swap* or *load-locked* and *store-conditional* operations is not needed, simplifying portability.

In the case that a synchronized Java method or statement does not contain a *synchronization point*, it is not even necessary to enter or exit the monitor. It is sufficient to test if the monitor is available, and wait for it to become available if this is not the case. Since other threads can become active only at *synchronization points*, no other thread can possibly request this monitor before it would be released by the current thread, so no code for entering or exiting it is required here.

A very efficient monitor implementation for Java is presented in [YLPMEA99], with a reported performance increase of a factor of 21. But the presented mechanism does not work with native threads, it requires user-level threads with explicit triggering of thread scheduling. With the automatic creation of *synchronization points*, this scheme can be used without explicit scheduling by the Java program, as if the system would use native threads.

Dealing with native code

A particular problem arises with the presence of native code, as it might be used in Java using the Java Native Interface Specification [JNI97]. Since a compiler or virtual machine has no influence on the native code, it cannot be guaranteed that code for *synchronization points* is being executed sufficiently often. Native code that waits for some I/O event or that tries to obtain a monitor might force the whole system to wait or even cause a deadlock.

For Java, the JNI interface has been designed to disallow any direct accesses to the Java memory within native code. Any accesses have to be performed through calls to routines of the JNI interface. This allows us to safely run native code in a native thread that is not synchronized with other Java threads or the garbage collector. To perform the call, we have to record the current state of the root set and allow other Java-threads to become active while the native code is running. When the native code returns, we have to ensure that other Java-threads are stopped at a *synchronization point* before we can continue normal execution. **Figure 9** illustrates the required code.

```
...store information on thread's state...

// allow for thread switches:
V(global_thread_semaphore);

// execute the native code while other java threads
// are allowed to run
Native_Call();

// disallow thread switches
P(global_thread_semaphore);
```

Figure 8: pseudo-C code to call native method from Java code

Field packing

Current Java implementations use an object layout that allocates at least one machine word of memory for every field, even though several smaller values like *byte* or *char* types could be stored within a single word. One reason for this is that on modern RISC architectures like Alpha [Sites92], no atomic operations to write subword values are present. To write a byte it is required to first load the word containing the byte that is to be overwritten, insert the new value at the desired position and write the modified word back to memory.

Without any further synchronization and with the presence of native threads, this is incompatible with Java's concurrency model, since a concurrent write of a different field, that happens to be held in the same word, might have no effect.

Again, the presence of *synchronization points* removes this problems since the sequence of instructions to write sub-word data automatically becomes atomic. The denser object model that becomes possible will reduce memory demand and at the same time increase locality.

Experimental Results

To get a better impression on the costs and gains of the presented mechanism, we have collected information during execution of three large Java applications during runtime. To record this information, the applications were compiled with an extended version of the TurboJ Java-bytecode compiler [TurboJ].

The applications that were examined are

1. TurboJ [TurboJ], a highly optimizing Java bytecode compiler. We recorded its behaviour while compiling all 1610 Java standard classes.
2. SwingSet [SwingSet], the demo application that comes with the Swing GUI class library and that makes extensive use of the classes in this library.
3. HotJava [HotJava], a Web browser written in Java. During the test we used it to browse the Web pages of The Open Group Research Institute [TOG-RI].

To facilitate reproduction of our results, we disabled inlining, since the aggressiveness of inlining has an important impact on other optimizations.

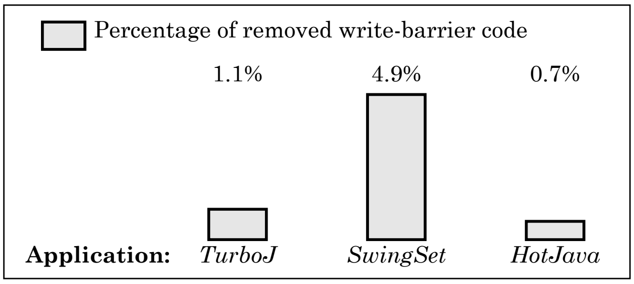

Figure 9: Percentage of write-barrier code that could be optimized

First, we have added an instruction for the marking of the assigned reference in a typical write-barrier. For this instruction, we allowed simple common subexpression removal optimizations limited to extended basic blocks without intermediate *synchronization points*. During execution we counted the number or marks that were executed and those that could be optimized. **Figure 9** illustrates the results: For TurboJ and HotJava, this optimization is insignificant, while in SwingSet nearly 5% of the write-barrier code could be removed. Allowing inlining provides more opportunities for optimization, and it can be expected that a better optimization algorithm that is not limited to extended basic blocks will also enhance the results.

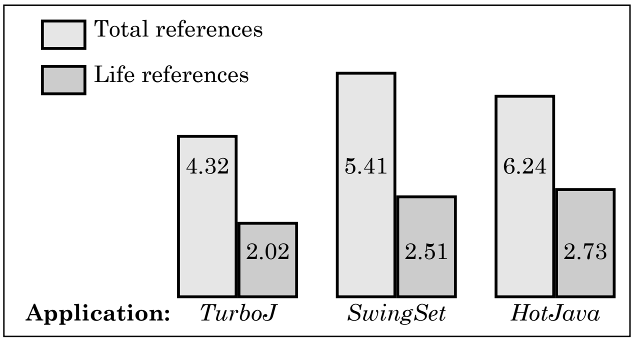

Figure 10: Average number of total and life reference variables on a call

Next, we measured the overhead for constant-time root scanning as presented above. To do this, we counted the number of local variables of reference types (these include local variables found in the Java bytecode, local variables to emulate the Java stack and temporary variables introduced by the compiler for different purposes) that exist at each call point. We have recorded the total number of such variables, and the number of variables that are life at the call point. The results of this simulation are present in **Figure 10**, given as average numbers of reference local variables at the execution of a call. The average number of total references present at a call varies from 4.32 to 6.24, while the number of life references is only between 2.02 and 2.73. The additional overhead of copying these references to the heap on every call corresponds to just a few heap stores per call, while saving only the life variables has about half the overhead of saving all references. Since life values have to be stored in a caller-safe place anyway, the additional overhead appears easily acceptable, especially when regarding the benefits of removing GC latencies due to root scanning completely and avoiding the the need to provide any further information on root variables.

Conclusion

We have presented a new way to synchronize mutator threads and garbage collection on single processor systems. We have shown that this allows to provide information for exact incremental garbage collection that is hard to achieve when native threads are used directly. Furthermore, we have shown that the presented mechanism allows more efficient code by avoiding the need for atomic code sequences and by enabling write-barrier code to profit from standard optimizations such that the compiler might even remove write-barrier code completely for some writes.

We have shown that several optimizations are possible that can reduce the introduced overhead to a minimum, and that a trade-off between this overhead and the delay for pre-emption can be made.

Finally, we have presented ways to deal with three specific problems and applications when using the presented mechanism in an implementation of Java: monitors, native code and field packing.

Since we have limited our analysis on the synchronization aspects, for a complete real-time garbage collector implementation many other decisions have to be made. A suitable incremental algorithm has to be chosen, among the classical ones are [Dijkstra78], [Baker78] and [Baker92]. The algorithm has to be implemented carefully so that it is exact and does have a sufficient worst case performance behaviour (many current GC implementation have a worst case performance in $O(n^2)$ for a heap size of n words, which is unacceptable in real-time systems). Then, a mechanism that guarantees sufficient GC progress has to be used. Examples of such mechanisms using fixed or flexible GC rates on allocation can be found in [Wilson98] and [Siebert98]. Another problem that needs to be addressed is fragmentation, since a system that might fail due to fragmented memory is just as useless as one that cannot satisfy its real-time constraints. The implementation must either defragment memory by moving objects or use a different means to limit the worst case amount of memory lost due to fragmentation.

We think that efficient and reliable garbage collection that gives hard real-time guarantees should be standard today. We have shown that the information required for such a garbage collector can be provided with reasonable run-time overhead using the presented mechanisms.

References

[AG98] Ken Arnold and James Gosling: The Java Programming Language, 2nd edition, Addison Wesley, 1998

[Baker78] Henry G. Baker: *List processing in Real Time on a Serial Computer*. Communications of the ACM 21,4 (April 1978), p. 280-294, ftp://ftp.netcom.com/pub/hb/hbaker/RealTimeGC.html

[Baker92] Henry G. Baker: *The Treadmill: Real-Time Garbage Collection without Motion Sickness*, ACM SIGPLAN Notices 27,3, 1992, pp. 66-70

[Barlett88] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: *Thin Locks: Featherweight Synchronization for Java*. ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998

[BLT98] Philippe Bernadat, Dan Lambright, Franco Travostino: *Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments*. IEEE Workshop on Programming Languages for Real-Time Industrial Applications, Madrid, 1998

[Dijkstra65] Edsgar W. Dijkstra: *Cooperating Sequential Processes*, Technical Report EWD-123, Technical University Eindhoven, 1965

[Dijkstra78] 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

[DG94] Damien Doligez, Georges Gonthier: *Portable, Unobtrusive Garbage Collection for Multiprocessor Systems*, ACM Symposium on Principles of Programming Languages (POPL), 1994

[HKHW96] Steve Hoxey, Faraydon Karim, Bill Hay, Hank Warren: *The PowerPC CompilerWriter's Guide*, IBM Microelectronics Division, New York, 1996

[HotJava] *HotJava (tm) Browser*, V1.1.4, 1998, Sun Microsystems Inc., http://www.javasoft.com/products/hotjava/1.1.4/

[HW98] Lorenz Huelsbergen, Phil Winterbottom: *Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization*, International Symposium on Memory Management, 1998

[Inmos93] *The T9000 Transputer Instruction Set Manual*, INMOS Transputer book series, INMOS Ltd., SGS-Thomson Microelectronics Group, 1993

[JDK] *Java Development Kit 1.1.x*, SUN Microsystems Inc., 1999 http://www.javasoft.com/products/jdk/1.1

[JNI97] *Java Native Interface Specification*, SUN Microsystems Inc., 1997

[KR94] Jens Knoop, Oliver Rüthing: *Optimal Code motion: Theory and Practice*, ACM Transaction on Programming Languages and Systems, Vol. 16, No. 4, July 1994, pp 1117-1155

[LY96] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification*, Addison-Wesley, 1996

[Nilsen94] Kelvin Nilsen: *Reliable Real-Time Garbage Collection of C++*, Computing Systems, Vol 7 no. 4, 1994

[Meyer92] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall International (UK) Ltd, Hertforshire, 1992

[Pirinen98] Pekka P. Pirinen: *Barrier techniques for Incremental Tracing*, International Symposium on Memory Management, 1998

[Siebert98] Fridtjof Siebert: *Guaranteeing non-disruptiveness and real-time Deadlines in an Incremental Garbage Collector*, International Symposium on Memory Management, 1998

[Sites92] Richard L. Sites: *Alpha APX Architecture*, Digital Technical Journal Vol. 4 No. 4 Special Issue, 1992

[SwingSet] The *examples/SwingSet* program of the Swing 1.0.1 class library, SUN Microsystems Inc., 1998, http://www.javasoft.com/products/jfc/

[TOG-RI] The Open Group Research Institute, Grenoble, France. http://www.gr.opengroup.org/

[TurboJ] *TurboJ V1.1.2*, ahead-of-time Java compiler, The Open Group Research Institute, Grenoble, France, 1997-1999 http://www.gr.opengroup.org/openitsol/

[Wilson98] Paul R. Wilson: *Uniprocessor Garbage Collection Techniques*. Submitted to the ACM Computing Surveys. 1998

[YLPMEA99] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioğlu, Erik Altmann: *Lightweight Monitor for Java VM*, ACM Computer Architecture News, Vol 27-1, March 1999