Real-Time Non-Copying Garbage Collection*

Paul R. Wilson and Mark S. Johnstone
Department of Computer Sciences
University of Texas at Austin
wilson,johnston@cs.utexas.edu

1 Overview

Real-time garbage collection is not popular, despite its obvious attractiveness for many applications. This is largely due to the perceived cost and disruptiveness of garbage collection in general and incremental garbage collection in particular.

Most existing "real-time" garbage collectors are not in fact usefully real-time, largely due to the use of a *read barrier* to trigger incremental copying of data structures being traversed by the running application. (The read barrier consists of extra instructions accompanying each pointer use, which ensure that the running program will never see inconsistencies due to encountering partially-copied data structures.) This may slow down running applications unpredictably, even though individual increments of garbage collection work are small and bounded.

We have developed a hard real-time garbage collector for C++ which uses a *write barrier*, only coordinating collection work with *modifications* of pointers in data structures, making coordination costs cheaper and more predictable. We combine this write barrier approach with *implicit non-copying reclamation*, which has most of the advantages of copying collection (notably the avoidance of the sweep phase required by mark-sweep collectors, and and the touching of garbage objects when reclaiming their space).

Because a non-copying collector cannot compact live data, fragmentation is a potential problem in our system. We believe that the problem is not as severe as it appears at first glance, however, and that for most programs, usable worst-case space bounds can be ensured. For other programs, it may also be feasible to rely on an occasional compacting collection with some degradation of real-time guarantees.

*Position paper for the 1993 ACM OOPSLA Workshop on Memory Management and Garbage Collection.

This implementation strategy is well suited to stock hardware and operating systems. Because the write barrier records the same information as the write barrier for a generational scheme, our collector can easily be made generational to provide excellent average-case efficiency while preserving hard real-time response. We also believe that our collector is easier to parallelize than copying collectors.

2 Real-Time Collection: What it is and When it Isn't

Real-time garbage collection must be *incremental*; that is, it must be possible to perform small units of garbage collection work while an application is executing, rather than halting the application and performing large amounts of work without interruption. Strict bounds on individual garbage collection pauses are often used as the criterion for real-time garbage collection, but for practical applications, the requirements are often stricter still. The application must be able to make significant *progress*. That is, for a garbage collector to be usefully real-time not only must the pauses be short and bounded, they must not occur too often. The garbage collector must be able to guarantee not only that every garbage collection pause is bounded, but that for any given increment of computation, a minimum amount of the CPU is always available for the running application.

Baker's incremental copying technique is the best-known real-time collection strategy, but it is actually poorly suited to real-time garbage collection on stock hardware; its close coupling between application program actions and collector actions makes it intrinsically more expensive and difficult to use for real-time applications. Even though any given pause caused by the collector is short and strictly bounded, pauses may be clustered closely together causing the application to miss its larger granularity deadlines.

Baker uses a *read barrier* (special code potentially executed at every pointer reference) to maintain consistency between the running program and the garbage collector. Thus, every reference to a pointer potentially causes an increment of garbage collection to be performed. The unfortunate act of traversing a list that has not yet been reached by the collector can cause all of the objects in that list to be copied. While copying each object takes a strictly bounded amount of time, copying the entire list can keep the application from getting a reasonable fraction of the CPU's time, making it miss its real-time deadlines. This *read barrier* cost is potentially high, and very unpredictable, because the cost of traversing an ordinary list is strongly dependent on whether the list has already been reached and copied by the collector [Nil88, EV, Wit92].

This problem is even worse in recent collectors which use pagewise virtual memory protections to trigger larger increments of collector work [AEL88, Det90, Joh92], but it is also significant on Lisp-machine style hardware. Even if the necessary checks are performed by dedicated parallel hardware, most of the available CPU time may be used up (in the worst case) by the actual trapping to copying routines and the copying itself. Complex memory subsystem hardware could solve this problem [NS90], but such hardware has not yet been built and is unlikely to be widely available (or economical) soon.

Nilsen and Schmidt [NS90] argue that even if increments of garbage collection work are small, a real-time program may miss its deadlines if too many small increments add up to too much total overhead over some period of time relevant to a deadline. For example, a collector might impose 30 instructions overhead per pointer dereference in the worst case. If the program attempts to execute a very high percentage of pointer dereferences, it may spend the majority of its time doing garbage collection work, and miss one of its deadlines.

Nilsen's solution to this problem is to require that the worst-case delays for any program operation be small relative to that operation's normal execution time [NS90]. So, for example, if the worst case for any operation is only twice the usual case, there is no possibility for any increment of computation taking more than twice as long as it would without garbage collection.

We believe that this is unnecessarily restrictive. To meet a real-time deadline, it is only necessary that the garbage collector not use up too large a fraction of the available CPU cycles *at a timescale relevant to the program's deadlines.* Our strategy is therefore to allow the collector to incur dozens of instructions of overhead on some pointer operations, so long as we can guarantee that this will not happen too often. (Naturally, "too often" must be quantified relative to the application's responsiveness requirements.) Such a guarantee requires a weaker coupling between program operations and collector operations than copying collectors generally provide.

3 Non-Copying, Write-Barrier Incremental Techniques

We have revived incremental techniques designed in the mid-1970's for (non-real-time) concurrent garbage collection, and applied them to a real-time collector for stock uniprocessors. Coordination between the running application and the collector's tracing traversal is via a *write barrier*, i.e., the collector's view of data structures is updated only when the application *modifies* the graph of pointer relationships. Compared to Baker's *read barrier* technique, which coordinates the collector with the application whenever the application reads or compares pointers, this reduces coordination costs and, equally important, makes them much more predictable.

Write-barrier algorithms coordinate the collector's view of data structures with the running application's view—whenever the application modifies a data structure by changing a pointer, the collector must be protected from being confused and "losing" objects. This can happen if an object is hidden from the collector by storing a pointer to the object into another object that the collector has already examined, and then breaking all other paths to the object. If a pointer from an object that the collector has already traced is the only remaining path to another object, that object will never be found—it may be erroneously reclaimed, and its space reused, leaving a dangling pointer that the application could still traverse.

This problem is best understood via the abstraction of *tricolor marking* [DLM+78]. At the beginning of a garbage collection cycle, we consider all objects to be *white*. When the traversal reaches an object, it is colored *grey*. When a grey object's pointer fields are scanned and their referents are greyed, it is colored *black*. The significance of these three colors is:

- White objects are those that have not yet been reached by the collector's tracing traversal. If at the end of collection an object is still marked white, then it is known to be garbage.
- Gray objects are those currently under consideration by the collector, because they are known to be reachable, but it is not yet known what other objects are reachable from them. In practical terms, this just means that the objects are in the stack (or queue) that controls the collector's traversal of reachable data.
- Black objects are those that the collector is through with—they have already been examined and their role in the reachability graph is known. This generally means that they have been removed from the traversal stack (or queue).

The main problem of non-copying incremental collection is to ensure that the collector's notion of the reachability graph does not get out of synch with the actual reachability graph, due to changes made by the running program. If the program stores a pointer to a white object into a black object, and nothing special is done, the pointer will not be found by the collector (Recall that the garbage collector has already examined black objects, and will not look at them again.) If all other paths to the white object are broken before being reached by the collector, the white object will be reclaimed. Since the application can still reach the white object through the black object's pointer, this creates a dangling pointer.

To prevent this, two basic strategies are possible [Wil92]. One strategy, *snapshot-at-beginning*, is to ensure that objects can never get lost, by preventing any pointers from being destroyed [AP87, Yua90]. Before overwriting a pointer, the old pointer value is saved away so that the collector can still find it. We call this a snapshot-at-beginning algorithm because the collector's view of reachable data structures is fixed when collection begins.

The other strategy for collection, *incremental update*, focuses on the writing of new pointers into objects that the collector has already reached and examined. When such a pointer is created, the collector is notified so that it can either trace the pointed-to object immediately, or re-examine the stored-into location again later, to find any "hidden" objects [Ste75, DLM+78, BDS91]. That is, the collector's view of reachable data structures is incrementally updated in the face of changes to those data structures by the running program.

Our collector uses an incremental update strategy for four major reasons:

- Adaptability to generational collection: The write barrier used for incremental collection can serve double duty as a write barrier for tracking inter-generational pointers in a generational collector [Wil92].[1]
- Reduced conservatism: Snapshot-at-beginning algorithms retain all objects that are live at the moment incremental collection begins, and objects that become garbage during collection are retained until the end of the next garbage collection cycle. Incremental update algorithms are less subject to this "floating garbage," because they can can reclaim many objects that die before being reached by the collector.[2]
- Potential for optimization: We believe it is easier for smart compilers to optimize this kind of write barrier, because it preserves a local constraint rather than a global one.
- Suitability for hierarchical and multistrategy garbage collection: An important form of garbage collector optimization is the use of *hierarchical* collection strategies, which combine fast (and somewhat conservative) collection of small areas with slower (but more comprehensive) collection of larger areas. Similar arguments apply to combinations of manual and automatic deallocation.

[1] A collector for a purely real-time system does not gain much from generational techniques, because they do not typically improve worst-case performance. For some applications, however, it may be possible to provide sufficient guarantees about lifetime distributions to support true real-time generational collection. Systems with mixed hard and soft deadlines may also benefit considerably, however, because generational techniques can greatly improve average-case performance.

[2] We intend to enhance this by using *opportunistic traversal* strategies, which heuristically attempt to traverse relatively stable data structures first.

3.1 Snapshot-at-Beginning Algorithms

Snapshot–at–beginning algorithms preserve a global constraint: if a path to an object is broken, another path to that object will be created. The write barrier simply saves any overwritten pointer. This means that pointers directly from black objects to white ones can be created, and the collector only guarantees that there will *also* be another path to the object that it *will* traverse—i.e., a path through some grey object somewhere.

The global nature of this constraint has deep implications. The fact that an object is grey signifies more than the existence of a path through it to reachable objects. In effect, a grey object may represent the possibility of many other paths, elsewhere in the heap. This possibility makes it difficult to perform local optimizations of the garbage collector. For example, if a pointer disappears from the stack (due to being overwritten, or the popping of a stack frame), it must be retained and traversed, because it might be the only pointer not from a black object, to a set of live objects which must not be reclaimed. This inhibits any special-purpose treatment of the stack, which is important for good performance in many systems.

Another problem is that if liveness information is available from another source—such as a complementary garbage collector, or explicit deallocation by the programmer—that information cannot be used to short-circuit the garbage collector's traversal. The fact that an object is known to be garbage only means that its space can be reclaimed—but not without traversing its pointers, because they may implicitly represent other, unnoticed pointers from black to white objects, elsewhere in the heap.

3.2 Incremental Update Algorithms

Incremental update algorithms are quite different from snapshot algorithms, because they preserve a local constraint—no black object is allowed to hold a pointer directly to a white object. If a pointer to a white object is stored into a black object, the white object must be immediately greyed (added to the collector's traversal queue), or the black object must be reverted to grey (i.e., put back in the queue so that it will be re-examined later). This ensures that the garbage collector will encounter every pointer from an object that has been reached to an object that has not.

We refer to this local constraint (no pointers directly from black to white objects) as the *tricolor invariant*. The tricolor invariant is much less conservative, in terms of what is considered reachable, than the global constraint of a snapshot strategy. If pointers between white objects are broken, the garbage collector need not take them into account. If the white objects become unreachable from any grey objects, so much the better—they are garbage anyway and shouldn't be traversed. It is therefore possible to reclaim some objects that become garbage during collection.

The tricolor invariant also allows more optimization than is possible with a snapshot strategy. The stack can be considered white (unreached) up until the last moment during garbage collection, so that the creation and destruction of pointers from the stack can be ignored until then. This avoids considerable write barrier overhead for pointer stores into stack-allocated variables, and should allow more short-lived objects to be reclaimed.

Still another optimization allowed by the local constraint is that the collector can use any available oracle which can tell it if a white object is already garbage. For example, in combining garbage collection of some objects with explicit deallocation of others, the programmer may notify the collector that an object will never be used again beyond a particular point in the program's execution. If the object is white or grey, the collector can use that information to short-circuit its traversal at that object. Assuming the oracle is correct, this will not erroneously reclaim any reachable objects—if there are other paths to other white objects, there will be grey objects on those other paths, and the collector will find them. This kind of optimization is also the essence of hierarchical garbage collection, which we will discuss later.

4 Nonmoving Implicit Reclamation

We have combined an incremental update write barrier with a generalization of Baker's non-copying *implicit reclamation* strategy, so that unreached objects needn't be traversed to be reclaimed, as they are in the sweep phase of a mark-sweep collector [Bak91].[3] A copying collector copies the live data out of a region of memory,

[3] It appears that this same basic strategy was used in a non-incremental collector for C++ by Thomas Wang, and is described in a Masters Thesis from California Polytechnic at San Luis Obispo; we have not yet seen that thesis, however.

and can thus reclaim that region of memory in its entirety without examining it, i.e., implicitly. It therefore avoids the cost of a sweep phase, which must scan through memory to find unreached objects and reclaim their space.

A non-copying implicit-reclamation collector achieves the same effect by maintaining data structures that record which *set* objects are in, and "moving" an object from one set to another rather than literally copying them from one area of memory to another. These sets are implemented as doubly-linked lists, plus a header field in each object saying which set it resides in. As the reachable objects are traversed by the collector, they are unlinked from the old list of live objects and linked into a new one. After all reachable data have been moved from one list to another, the remaining objects on the old list are known to be garbage, and that list can simply be appended to the free list in small constant time. This pseudo-copying is cheaper than real-copying for most objects; it also avoids the need to keep the relocation of objects from confusing the running application.[4]

Our collector generalizes this scheme by combining it with a *segregated storage* scheme for the management of different-sized chunks of memory. Separate sets of lists are used to manage different-sized objects, with objects of similar sizes grouped into "size classes."

Currently, our size classes are powers of two. When allocating an object, we round its size up to the next power of two and allocate it in that size class. Like any noncompacting storage scheme, our segregated storage scheme is vulnerable to fragmentation, but the worst-case fragmentation is bounded. We expect the fragmentation cost to be low for most programs, and we decrease the worst-case bound (at a cost in average-case fragmentation) by using relatively coarse gradations of sizes. This keeps memory from ever being divided too finely and hence becomming unusable for different-sized objects. The current scheme does no coalescing of adjacent freed blocks into larger free blocks.

One complication of using segregated storage is that we cannot count on ever exhausting a given free list, as Baker does; that is, free chunks may remain on a free list for an arbitrary number of collection cycles. This means that when a set of objects is reclaimed, we cannot simply change the meaning of the color bit patterns in the object's header as Baker describes; we must actually change the color field of each object that we reclaim. While this is not as elegant as Baker's scheme, because it introduces a cost proportional to the number of objects reclaimed, in practice it doesn't really matter much because this cost is very small.

To keep from having to reset the color fields of every reclaimed object when we reclaim a set of garbage objects, we perform this recoloring lazily. We append the garbage objects to the list of free objects, so that the freeset contains objects of multiple colors, and we wait to reset the headers until the objects are reallocated[5]. Since neither the application nor the collector examines the objects in the free set until they are reallocated, this works just as well, and spreads the cost over the allocations, rather than incurring it all at once.

5 Interface to C++

Currently, our collector uses a *smart pointer* interface to C++; garbage collected objects have an associated pointer type defined in a library as a parameterized class, and client code must use these pointers rather than raw C++ pointers. (Parameterization and operator overloading make this relatively easy, although smart pointers can not be used quite as flexibly as raw pointers [Ede92].) The main difference between our parameterized pointers and normal pointers is that pointer assignments execute an additional few lines of code, which constitute the write barrier.

In C++, pointers may point to an element of an array or a substructure of a record. In addition, because of the usual C++ implementation strategy for multiple inheritance, a pointer may also point to a subcomponent of an object whose layout is a concatenation of the layouts of classes from which it is derived. Because we use a segregated storage scheme, with all objects aligned on known word boundaries, it is relatively easy to derive the start of objects from these interior pointers.

Unfortunately, recovering object headers from derived pointers is still a major source of overhead,

[4] The main motivation for Baker's read barrier is really to keep the running application from seeing temporary inconsistencies in data structures while they are being copied by the collector—that is, the read barrier protects the application program from changes made by the collector, as well as keeping the collector from being confused by the application's changes.

[5] Conceptually, we recolor all of the freed chunks of memory at the instant a collection is complete, but the chunks' color fields are only reset when they are reused to allocate new objects.

because it slows down the write barrier. If the compiler or programmer can declare that a pointer will always point to the beginning of an object (or some fixed offset into it), much of this cost can be optimized away. In many languages this is trivial, and it appears that this optimization is easy for a C++ compiler in many common cases.

In our system, each object has a hidden header field, created by our overloaded version of the C++ `new` operator. The actual type descriptor information is constructed by compiling the program with the debugging option turned on, and using a special program which extracts structure layouts from the debugging information. (We use code from the GNU `gdb` debugger for this, so our collector should be easily portable to any system that uses a debugging output format that `gdb` understands.)

With compiler cooperation our collector would be trivial to use, and more efficient than the current smart-pointer version. While we currently use our collector for C++, it could easily be adapted for use with any garbage-collected programming language. We have recently ported the collector to our new Scheme system, and implementors of Eiffel and Dylan have also expressed an interest in using it.

6 Coping with Fragmentation

For reasons of efficiency and real-time bounds, we have chosen a non-copying collection strategy. This brings with it the problem of fragmentation, since in general we cannot ensure that freed memory is contiguous. In the worst case, our memory usage is extremely bad, but we expect that in practice, memory usage will be much better. Equally important, we believe that memory usage will be provably acceptable for most programs.

Consider the special case of a program with only one size of heap-allocated object. In that case, our system's memory usage is very similar to that a of a copying collector, in that any freed memory can be used to satisfy any allocation request. In another respect, however, memory usage in our system is better—because it is not a copying collector, no extra space is needed to hold the extra versions of objects being copied.

For programs with a variety of allocation request sizes, the memory usage of our system could deteriorate rapidly. In the worst case, a program might allocate exclusively from one size class for an entire garbage collection cycle, and then from another for the next garbage collection cycle, and so on. In that case, the worst-case memory usage is proportional to the product of the maximum amount of live data times the number of sizes managed. This would be a very bad case indeed.

In practice, most programs allocate large numbers of only a few sizes of objects, and we believe that acceptable worst-case bounds for particular programs will usually be easy to derive. For many other programs, it may be possible to alleviate potential fragmentation problems by strategies such as fragmenting some objects into smaller-sized chunks.

For still other systems, it may be acceptable to augment our collector with a compacting collector that cannot satisfy the same time bounds. (We are interested in using something like the Appel-Ellis-Li pagewise copying scheme for a distributed persistent store, with our non-copying collector providing hard real-time performance for programs that only access memory-resident data.)

7 Future Work

Aside from tuning our collector and gaining experience with its space and time bounds, we intend to make several enhancements. It's current design is largely motivated by ease of extension to generational, distributed, and parallel collection.

We intend to make the collector generational in what we refer to as a truly *hierarchical* fashion, with fast small-scale collections nested within slow large-scale collections. Most if not all existing generational incremental collectors do not allow multiple ongoing collections at the same time. For example, in Moon's collector for the Symbolics Lisp Machines, only one collection may be ongoing at any time, although it may include more than one generation. (Most collections collect only the youngest generation, but some collect several generations simultaneously.) We see a problem with this common scheme for real-time and near-real-time applications. Large-scale collections should proceed slowly over long periods of time, with small-scale collections proceeding nearly independently. This will allow space to be reclaimed and reused quickly, even while a large-scale collection is in progress, without a major degradation in performance.[6]

[6]In contrast, systems which allow only one ongoing collection must suffer systematic performance degradations

Similarly, we intend to extend our algorithm for use in hierarchical distributed garbage collection.

8 Conclusions

We have implemented a hard real-time tracing collector for C++ which meets more stringent real-time requirements than any other implemented collector we know of, for either conventional or stock hardware. To our knowledge, this is the first actual implementation of an incremental non-copying implicit reclamation collector, and the first hard real-time incremental update collector. Our overall philosophy for real-time collecition is similar to that of Engelstad and Vandendorpe [EV], but our collector differs from theirs in several ways, most obviously in the use of implicit reclamation.[7]

Our initial, untuned implementation is relatively efficient, and most of the current cost (typically increasing run time by 10 to 90 percent) appears to be due to the lack of compiler cooperation rather than a flaw in the collector itself.[8] It already outperforms a deferred reference counting collector with a similar smart-pointer language interface [Det92]; we expect tuning to improve its performance considerably in the near future, and we will further improve it by making it generational and exploiting compiler cooperation.

Unlike deferred reference counting [DB76], our collection scheme can reclaim cyclic garbage. Unlike Baker's read-barrier algorithms (and also unlike coarse-grained algorithms relying on virtual memory cooperation), it provides reasonable hard real-time guarantees.

A generational version of our hard real-time collector should be competitive with existing state-of-the-art generational collectors in terms of raw performance. The compiler cooperation required for maximum performance is the same as that required for generational collection.[9]

Our collector is public domain, and is suitable for incorporation into any garbage collected language implementation. It requires no special hardware or operating system features.

when multi-generational collections traverse large amounts of live data. Since there are no independent younger-generation collections which can free up space quickly, the larger-scale collections must run fast to ensure that they terminate before the remaining free space is exhausted. This means that useful real-time performance is sacrificed during large-scale collections.

[7] We also believe that their generational collector fails to guarantee real-time response, and may even erroneously reclaim some objects, but those flaws could be fixed.

[8] Note that this overhead is measured in a high-performance C++ system. Performance would appear better in a relatively slower system, such as most Lisp or Smalltalk implementations.

[9] Because it is a non-copying collector, however, it can operate fairly efficiently with a lesser degree of cooperation, by using conservative pointer finding within stack frames [BW88, Boe93].

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. ACM Press.

[AP87] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier North-Holland.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. Position paper for OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems. Summary appears in [?], October 1991. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[Boe93] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1993. To appear.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[Det90] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.

[Det92] David L. Detlefs. Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference* [USE92].

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Ede92] Daniel Ross Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference* [USE92], pages 1–19. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992.

[EV] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real-time constraints. Position paper for 1991 OOPSLA Workshop on Garbage Collection in Object-Oriented Systems. (Available via anonymous internet ftp from cs.utexas.edu as pub/garbage/GC91/engelstad.ps.).

[Joh92] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1(1):46–58, March 1992.

[Nil88] Kelvin Nilsen. Garbage collection of strings and linked data structures in real time. *Software Practice and Experience*, 18(7):613–640, July 1988.

[NS90] Kelvin Nilsen and William J. Schmidt. A high-level overview of hardware assisted real-time garbage collection. Technical Report TR 90-18a, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1990.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[USE92] USENIX Association. *USENIX C++ Conference*, Portland, Oregon, August 1992.

[WH91] Paul R. Wilson and Barry Hayes. The 1991 OOPSLA Workshop on Garbage Collection in Object Oriented Systems (organizers' report). In Jerry L. Archibald, editor, *OOPSLA '91 Addendum to the Proceedings*, pages 63–71, Phoenix, Arizona, October 1991. ACM Press. Published as *OOPS Messenger 3*(4), October 1992.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[Wit92] P. T. Withington. How real is "real time" garbage collection? Position paper, OOPSLA '92 Workshop on Garbage Collection in Object-Oriented Systems, 1992.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.