

Real-Time Garbage Collection of Actors

By Douglas M. Washabaugh and Dennis Kafura

TR 90-24

Real-Time Garbage Collection of Actors

by

Douglas M. Washabaugh
Digital Equipment Corporation
tay2-2/b4
153 Taylor Street
Littleton, MA 01460
washabaugh@quiver.enet.dec.com

Dennis Kafura
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
kafura@vtopus.cs.vt.edu

ABSTRACT

This paper shows how to perform real-time automatic garbage collection of objects possessing their own thread of control. Beyond its interest as a novel real-time problem, the relevance of automatic management and concurrent objects to real-time applications is briefly discussed. The specific model of concurrent objects used in the paper is explained. A definition of real-time garbage collection is given and an algorithm satisfying this definition is described. An analysis of the relationship between latency, memory overhead and system size shows that this approach is immediately feasible for low-performance real-time systems or multiprocessor real-time systems with dedicated processor functions. Future improvements in the collector's performance are outlined.

1. Introduction

The overriding need for predictability [Stankovic 1988].has limited the software technology which is brought to bear upon real-time problems. Two such limitations are these. First, real-time developers have not availed themselves of the simplicity of automatic garbage collection as a memory management mechanism because almost all garbage collection algorithms exert their overhead at unpredictable times. Second, the lack of virtually any idea on how to perform automatic management of concurrent entities has not allowed the expressive power of small-grain concurrent to be exploited. Instead, only large-grain, programmer controlled forms of concurrency are used. To address both of these limitations we have developed a real-time automatic garbage collector for concurrent objects. Concurrent objects unify the abstractions of memory allocation and concurrency. The investigation of real-time garbage collection is part of a larger project whose long-term goal is to develop a new perspective on real-time systems based on small-grain concurrency, inheritance, and - the issues in this paper - automatic garbage collection of concurrent objects. The actor model [Agha 1986] is used as the framework for presenting the algorithms in this paper. However, the basic ideas apply to a larger class of concurrent object models.

The use of concurrent objects and automatic garbage collection is motivated by the following observations:

- autonomous, interacting real-world entities are more easily and more directly expressed in software as concurrent encapsulated objects with strictly limited interactions . This issue is explored further in [Kafura 1988].

- programmer controlled memory management is notoriously error-prone [Bloom 1987]. Failing to return a resource when it is no longer used and returning a resource that is still in use are mistakes are difficult to detect and repair. Furthermore, it is significantly more difficult for a programmer to correctly manage concurrent resources than traditional data resources..
- predictable operation of a garbage collection implies that it must be an integral part of the underlying automatic resource control system [Appel 1988a]. It is unlikely, or at least severely expensive, for each designer to attempt this task for each new application.





This work is also motivated by the deficiencies of existing collectors.

Related collectors are reviewed in Section 2 together with the actor model and a non-real-time garbage collector for actors. Section 3 defines a real-time garbage collector and shows how this definition is satisfied by extensions to the basic actor collector. Section 4 analyzes the memory required by the real-time collector under varying sizes of the actor system and varying latency constraints. Conclusions are given in Section 5.

2. Actors and Garbage Collection

The features of the actor model relevant to garbage collection are illustrated by an example. The notation used in this example is summarized in Table 1 and the example is shown in Figure 1.

Table 1. Legend for Actor Figures

Symbol	Interpretation of Symbol
	Blocked Actor
	Active Actor
	Root Actor
	Acquaintance Arc

The state of the message driven computation in an actor system can be depicted by a graph whose nodes represent actors and whose directed arcs represent acquaintances. An acquaintance arc from actor S to actor T means that, whenever S is active, S can send a message to T. Actors are either active (drawn as circles) or blocked (drawn as squares). An active actor may send mail messages asynchronously to its acquaintances and may also create new actors.

There is a set of transformations that can change an actor graph from a representation of what can *currently* happen to what can *potentially* happen. The two transformations relevant to the garbage collection problem are change in the state (active or blocked) of an individual actor and change in the topology of the system of actors. First, sending a message from an active actor to a blocked acquaintance allows the blocked actor to become active. For example, in Figure 1, if F send a message to G, G becomes active. Second, an active actor can send its own mail queue address or the mail queue address of one of its acquaintances to another of its acquaintances. This transformation changes the topology of

the actor system by introducing a new acquaintance arc. For example, in Figure 1, if B were to become active it could send the mail address of C to G, causing a new acquaintance arc to be introduced from G to C.

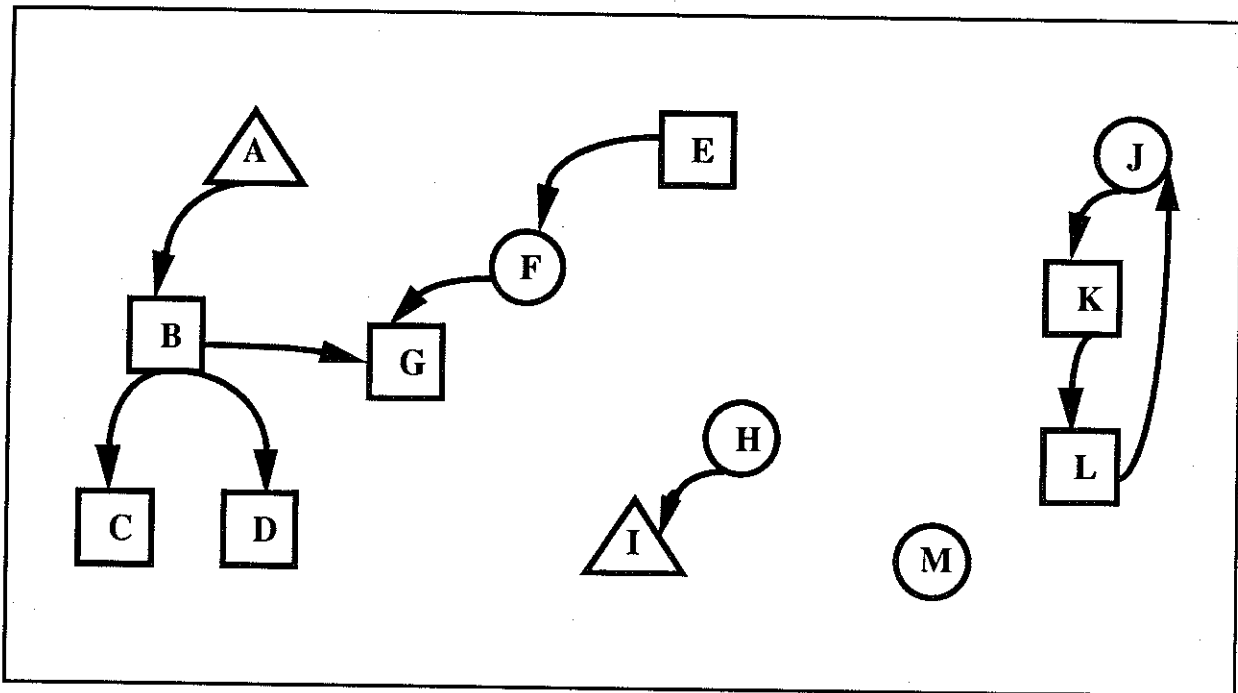


Figure 1. An Actor System

Informally, garbage actors are those whose presence or absence from the system cannot be detected by external observation excluding any visible effects due simply to the consumption of resources by garbage actors (e.g., increasing response time). To make this idea more concrete, a third type of actor, drawn as a triangle in Figure 1, is introduced. These actors are defined to be the "roots" of the actor system. Root actors are always consider to be active. Intuitively, root actors are the means by which the actor computation affects the "outside world." Root actors, for example, represent actuators or output ports.

Somewhat more precisely, a garbage actor is one which is:

1. not a root actor, and
2. cannot *potentially* receive a message from a root actor, and
3. cannot *potentially* send a message to a root actor.

In this definition, the term "potentially" refers to the results of applying the two transformations described above. A more comprehensive definition of garbage actors is given in [Washabaugh 1990].

One key property of garbage actors is that they cannot become non-garbage. Because actors are only determined to be garbage when there is no possibility of communication between it and a root actor, once an actor is marked as garbage, there is no possible sequence of transformations which would cause the garbage actor to become non-garbage.

Let us now consider which actors in Figure 1 are garbage. Actors A and I are root actors and by definition are not garbage. It can be easily seen that actors J,K,L and M are garbage - they cannot communicate with a root actor. Whatever actions they take cannot be made visible to the outside world. Notice that J and M are active while K and L are blocked. This shows that state alone is not a sufficient criterion for identifying garbage actors. Actor E is blocked and there is no way for it to become active because it is not the acquaintance of any other actor. However, actor H also is not the acquaintance of any other actor, but it is not garbage because it is active and can communicate directly with the root actor I. Message from the root actor A can reach actors B,C,D and G. If these messages contain A's mail queue address, these four actor can become active and communicate directly with a root actor. Hence, they are not garbage. Finally, actor F could send a message to G containing F's own mail queue address. G in turn could send A's mail queue address to F allowing F to communicate with the root actor A. So F is not garbage. This example illustrates that the relationships between the state of an actor and the current topology is complex.

Notice that if a simple marking algorithm is used for the system shown in Figure 1, actors E,F and H would be incorrectly marked as garbage because they are not reachable from a root. Also, simple reference counting can miss actors which are garbage. In Figure 1, actors J, K and L all have non-zero reference counts. Even though all of them are garbage they would not be considered as garbage by a reference counting scheme.

A Non-Real-Time Collector

The Push-Pull algorithm shown in Figure 2 implements the rules defined in [Nelson 1989] to "color" actors in a system. The algorithm uses three sets. Each set is named by a color with the following meanings:

- White Actors in this set are not necessarily reachable from a root actor.
- Gray Actors in this set are reachable from a root actor, but may not necessarily become active.
- Black Actors in this set are non-garbage. They are either root actors or are both reachable from a root actor and potentially active.

Every actor is always in exactly one of these three sets. The color of an actor is the color of the set of which it is currently an element.

The algorithm's name comes from its use of two coroutines: one coroutine "pushes" actors from the white and the gray sets into the black set, and the other coroutine "pulls" actors

from the white and gray sets into the black set. It is assumed that the mutator is halted while the coloring algorithm executes.

```
BEGIN Initialization  
  
All root actors are placed in the black set.  
All other actors are placed in the white set.  
Resume Puller  
  
END Initialization  
  
BEGIN Puller  
  
FOR [each actor in the black set not yet examined]  
    place non-black acquaintances of the actor in the black set  
END FOR  
resume Pusher  
  
END Puller  
  
BEGIN Pusher  
  
FOR [each actor in the white set]  
  
    CASE: actor is active and an acquaintance is black or gray  
        -> place actor in black set  
    CASE: actor is blocked and an acquaintance is black or gray  
        -> place actor in gray set  
  
END FOR  
  
IF [any actors were placed in the black or gray set]  
    THEN resume Puller  
    ELSE Termination  
  
END Pusher  
  
Termination:  
All actors which are not black are garbage
```

Figure 2. Push-Pull Algorithm

The actions of the Push-Pull algorithm are illustrated using the actor system shown in Figure 1. The initialization step puts actors A and I (root actors) in the black set and all other actors in the white set. For simplicity assume that actors are examined in alphabetical order. In the first pass, the Puller moves B into the black set since it is an acquaintance of A. A, B and I are now in the black set and A has been examined. The Puller next examines B and pulls its acquaintances (C,D and G) into the black set. The remaining elements in the black set do not have acquaintances so the Puller will finish without adding any other actors to the black set. The actors in the white set are now E,F,H,J,K,L and M. The Pusher will move F and H to the black set and leave all others unchanged. On pass 2 the Puller makes no changes while the Pusher moves E to the gray set (it is reachable but cannot become active). On pass three the Puller again takes no action. When the Puller also takes no action, it terminates. At the termination, the black set contains A, B, C, D, F, G, H, I. All other actors are garbage.

Related Garbage Collectors

Virtually all previous garbage collectors have focused exclusively on determining an object's "reachability" which, as was seen in Figure 1, is too weak a criteria for collecting active objects such as actors.

Baker and Hewitt [Baker 1977] described a variation of a mark-and-sweep garbage collector for functional languages which collected garbage expressions. Garbage collection in functional languages was also explored by Hudak [Hudak 1982] [Hudak 1983] who presented algorithms for marking a directed graph representing a distributed functional program. This work is relevant because of the close parallel between the concurrent evaluation of functional expressions and the concurrent execution of objects. However, there are basic differences between the functional model and the concurrent object model:

functional models do not have cyclic dependencies while concurrent objects may; concurrent objects do not evaluate to a single result as do functional expressions. Thus, the garbage collection techniques developed for functional languages are not directly applicable to object-based concurrent languages.

Halstead's garbage collector for distributed actors ([Halstead 1978]) uses the concept of an *actor reference tree*, which is a set of processors and connections between processors such that each processor has a reference to the actor. Garbage collection is performed by the reducing the actor reference tree until it contains a single processor. A local garbage collector is then used on each processor to collect garbage actors. A drawback of this method is that it cannot detect cyclic garbage. The algorithms presented in this paper collect cyclic garbage.

Emerald uses an object-based language for programming distributed applications [Black 1987]. Garbage collection in Emerald is discussed briefly in [Jul 1988]. From the standpoint of the actor model, the principle limitation of the Emerald approach is that it considers only reachability of objects. As indicated above this is too weak a criterion for collecting garbage in the actor model.

3. Real-Time Garbage Collection

In this section a definition of a real-time garbage collection is presented followed by a summary of Baker's [Baker 1978] real-time algorithm. An algorithm for the real-time garbage collection of actors is then presented as a series of extensions to the Push-Pull algorithm given in the preceding section. The analysis of this algorithm is presented in the next section.

Definition: The latency of a garbage collector is the maximum time during which the mutator must be suspended in order for the collector to perform any of its elementary operations.

Definition: A garbage collector is real-time if its latency is bounded by a constant independent of the number of cells in use [Baker 1978].

This definition implies that a real-time garbage collector must be incremental because performing an entire garbage collection while the mutator is halted is not acceptable. The definition also implies that the garbage collector must be concurrent, because the collector's thread of execution must be interwoven with that of the mutator. The next two subsections show how to achieve concurrent mutator-collector operation and incremental collection.

Mutator - Collector Concurrency

While a garbage collector is running, the mutator may:

- Create actors
- Create acquaintances
- Delete acquaintances

To guarantee that the garbage collector finishes in a finite amount of time, the collector must ignore actors or acquaintances created after the beginning of the collection. As shown in Figure 3, it is also necessary to ignore deletions of acquaintances that occur after the beginning of the collection. The top row of Figure 3 shows the addition the an

acquaintance (B to C) and the deletion of an acquaintance (A to C) over a period of time. At no time is actor C garbage. The second row of the figure shows the actor graph as it appears to a garbage collector that processes new deletions, but not new additions. This causes actor C to be incorrectly identified as garbage. Therefore, since changes by the mutator cannot be observed during a garbage collection, the garbage collector must work on a system that appears to be "frozen", i.e. in a consistent state.

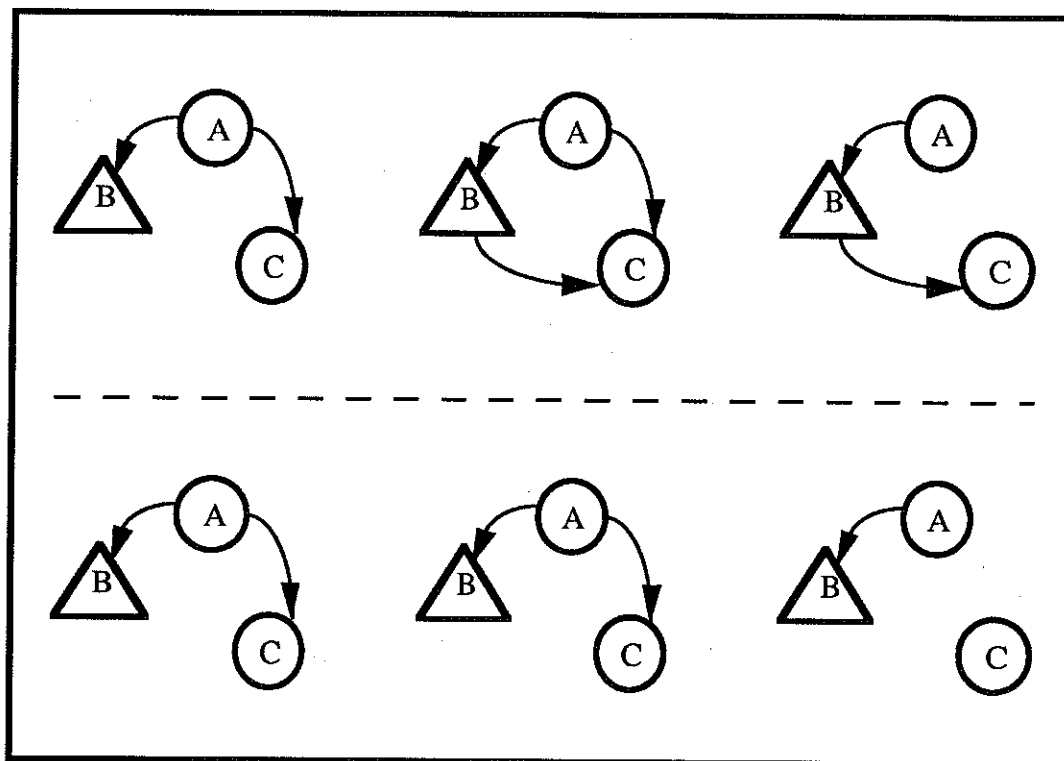


Figure 3 Additions and Deletions

There are two methods of achieving a consistent state without halting the mutator. One method, termed status marking, uses a single bit to indicate whether a mutator action (create or delete) was performed before or during a garbage collection. A second method

uses time stamps, called epoch numbers. Both methods permit the mutator to request that actions be done, but not to take affect, until a time convenient to the garbage collector.

Status marking is used to indicate whether a mutator action occurred before or during a garbage collection. Essentially, additions and deletions which occur after the initialization of the garbage collector are ignored until the collection is complete. When the mutator adds an actor or acquaintance, it marks it as new. If the mutator deletes an actor or acquaintance, it sets a delete_request flag. When the garbage collector starts, it first marks all actors and acquaintances as old. This marking may be done either atomically or non-atomically. If done non-atomically, then any changes that occur after the start of initialization are marked new, and therefore are not processed. When the garbage collector completes its status marking, it can delete all garbage actors and actors and acquaintances marked with delete_request. Note that any actors marked new are ignored by the current invocation of the collector. Figure 4 shows the interpretation of the status bits by the garbage collector.

Mark	Action
Old	Process actor or acquaintance
New	Ignore actor or acquaintance

Figure 4 Interpretation of Mark Bit

The advantage of status marking is that it requires only two extra bits, new and delete_request. The disadvantage is that it requires an extra pass over all actors and

acquaintances before and after the garbage collection. Therefore this method trades time complexity for space complexity.

The second method of obtaining a consistent state uses an epoch number. It is very similar to status marking, except that instead of using a single bit to indicate a relative time, a multi-bit value is used to indicate an absolute "time". The "time" is incremented at each invocation of the garbage collector. Thus, for the garbage collector to determine if an actor or acquaintance was created before or during a garbage collection, it compares the current epoch number to the epoch number associated with the actor or acquaintance. Whenever a garbage collection starts, it increments the epoch number. During the marking, the collector behaves in a manner identical to Table 3, with old defined as "object's epoch < current epoch" and new defined as "object's epoch = current epoch". These rules insure that the collector operates on a consistent system state. Because epoch numbers require more space than the two mark bits, this algorithm requires more data memory than the status marking algorithm, but an extra pass prior to the garbage collection is not required. Therefore this algorithm is more time efficient, but less space efficient than status marking.

Incremental Collection

The idea underlying incremental real-time garbage collection is that whenever a new resource is allocated, a constant amount of reclamation is performed. The amount of reclamation that is performed is chosen to guarantee that the mutator never depletes the resource pool. Baker's algorithm [Baker 1978] is an example of a real-time garbage collector for LISP systems based on the semi-space technique [Fenichel 1969]. It consists of a *from* semi-space, a *to* semi-space and a scavenger. The operation of Baker's algorithm is shown in Figure 5. Frame 1 shows the system just after a "flip" operation.

The *from* space contains all accessible cells, and the *to* space contains the root set. The root set, typically a bank of registers, is the set of cells from which all non-garbage nodes are reachable. Each cell consists of a car (pointer) and a cdr (data value). Every time that the mutator does a LISP "cons" operation (allocate a cell), a constant number of cells are scavenged. A cell is scavenged by examining the value of its car (address). If the car points to a cell in the *from* space that has not been copied into the *to* space, a copy operation is done. A copy operation makes a copy of the cell in the *to* space at the location pointed to by the copy pointer. The value of the copy pointer is written to the old cell in the *from* space as a forwarding address. This is done to tell the scavenger that a cell has already been copied, and tells the mutator where to find a cell that has been moved. Frame 2 of Figure 5 shows the two semi-spaces after the garbage collector has run for a period of time. The memory that is allocated from the new pointer is a result of mutator "cons" operations. Because the scavenge pointer is trailing the copy pointer, more cells still need to be reclaimed from the from space. In frame 3, the scavenge pointer has caught up to the copy pointer, signaling that the garbage collection is complete. A flip operation is done, which reverses the roles of the two semi-spaces. The result is shown in frame 4.

Baker's algorithm for real-time garbage collection cannot be applied to actors without adaptation because it interweaves the marking and copying of non-garbage cells. This approach does not work with actors because the marking of non-garbage actors is more complex.

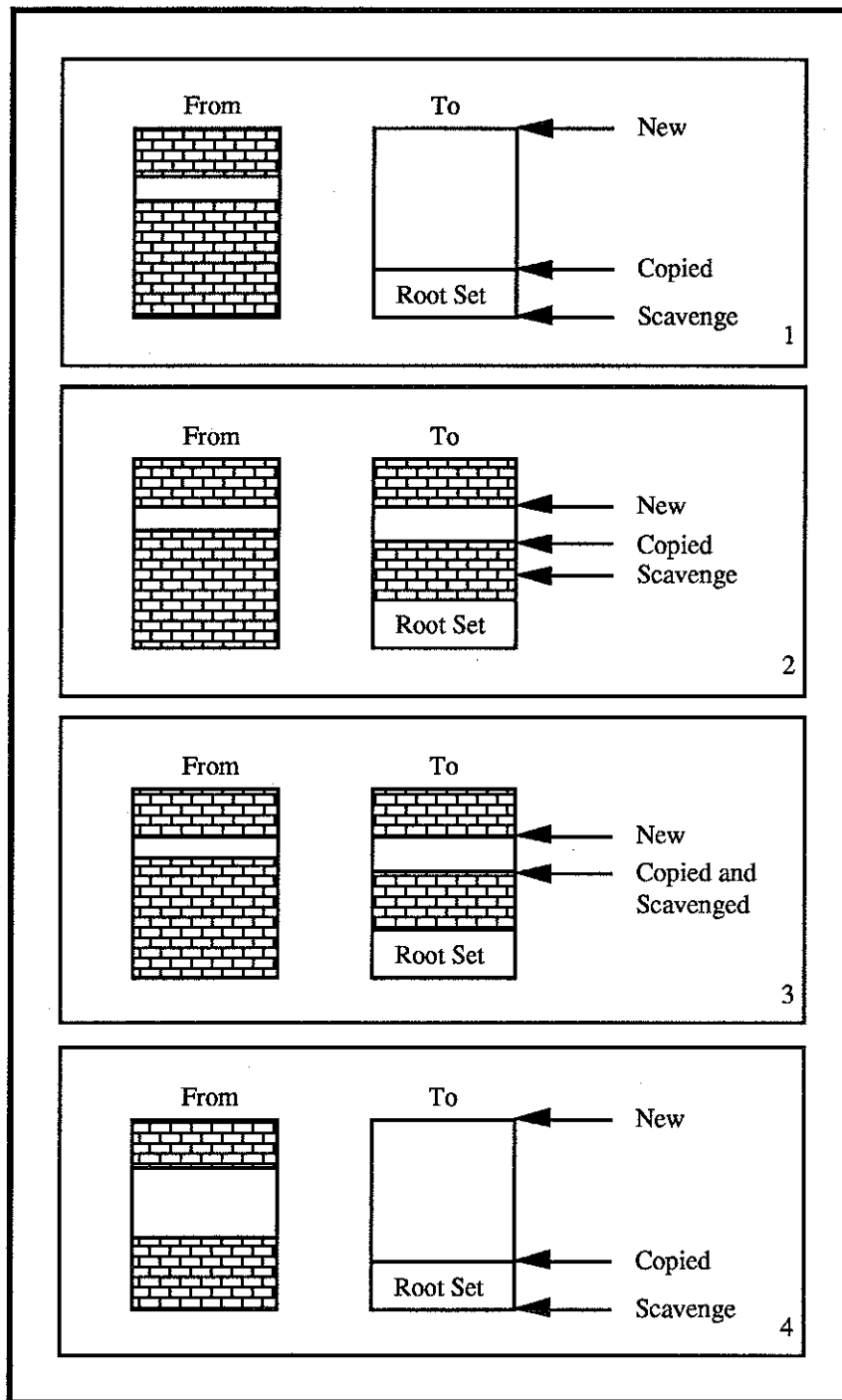


Figure 5: Baker's Algorithm

The remainder of this section describes the adaptations of Baker's algorithm needed to perform incremental collection of actors. Each time that an actor is created, the top level control structure of the garbage collector, shown in Figure 6, is executed. Its purpose is to perform an incremental amount of garbage collection. The routines `inc_mark` and `inc_heap_copy` are described in Figures 7 and 8, respectively.

```
BEGIN top_level_ctr
  IF [phase == mark] THEN
    inc_mark
    IF [marking finished] THEN phase = copy
  ELSE IF [phase == copy] THEN
    IF [heap copy not finished] THEN inc_heap_copy
    IF [heap copy AND code copy finished] THEN
      phase = mark
      set all actors with "garbage" bit to "free"
      inc_mark_init
    END IF
  END IF
END IF
END top_level_ctr
```

Figure 6 Top-Level Control

The incremental marker performs a bounded amount of marking on each invocation. Figure 7 shows the modified incremental push-pull algorithm. It uses the state variable `marking_phase` and an implicit actor pointer to save its place in the marking process.

```

BEGIN Inc_mark_init
    All root actors are placed in the black list
    All other actors are placed in the white list
    mark_phase = puller
END Inc_mark_init

BEGIN Inc_marker
    IF [mark_phase == puller] THEN -> Puller
    ELSE -> Pusher
END Inc_marker

BEGIN Puller
    mark_phase = puller
    FOR [next k actors in the black list]
        place non-black acquaintances of the actor at end of black list
    END FOR
    IF [end of black set reached] -> Pusher
END Puller

BEGIN Pusher
    FOR [next k actors in the white list]
        CASE: actor is active and an acquaintance is black or gray
            -> place actor at end of black list
        CASE: actor is blocked and an acquaintance is black or gray
            -> place actor at end of gray list
    END FOR
    IF [end of white list reached] THEN
        IF [any actors were placed in the black or gray lists]
            THEN mark_phase = puller
            ELSE done = true
        END IF
    END IF
END Pusher

```

Figure 7. Incremental Push-Pull Algorithm

```

BEGIN inc_heap_copy
    IF [scan_ptr == end of actor table] THEN                ; if all non-garbage copied
        flip                                                ; flip semi-spaces
        return "done"                                       ; done with collection
    ELSE IF [actor is non-garbage
              AND actor's heap segment is resident in from space
              AND actor's heap segment is not already relocated]
        do_not_schedule = TRUE                             ; actor can't run during copy
        allocate space by incrementing copy_ptr            ; allocate space
        copy heap segment to to space                      ; copy non-garbage code
        update heap base in actor table                    ; note relocation for actor
        do_not_schedule = FALSE                           ; actor can run now
    END IF
    update scan_ptr                                        ; move to next actor table entry
END inc_heap_copy

BEGIN create_new_heap_segment
    allocate space at new_ptr for heap                    ; allocate space in "to" space
    store size of heap in new heap segment                ; store size in heap
    store address of heap in actor table                  ; updated actor table entry
END create_new_heap_segment

BEGIN flip
    reverse roles of two semi-spaces
    new_ptr = location_0 in to space                      ; new pointer = start of space
    copy_ptr = location_m in to space                    ; copy pointer = end of space
    scan_ptr = location_0 in from space                  ; scan at start of linked list
END flip

```

Figure 8. Algorithm for Incremental Copy of Heap Segments

Because there is only one heap segment per actor, a heap segment becomes garbage when the actor which owns it becomes garbage. The collection of heap segments is also based on an incremental semi-space algorithm. Heaps are copied in their entirety, rather than in portions. Figure 8 shows the algorithm for the heap semi-space copy.

The reason that the actor which owns a heap segment cannot be scheduled while it is being copied is that heaps are writeable and inconsistencies result if locations that have been copied are modified. Code segments, which are read-only, do not have this problem. Baker's algorithm permits the mutator to access the semi-space during the copy, but requires the use of forwarding addresses.

4 Analysis of Latency and Memory Requirements

The garbage collector presented in the previous section fulfills our definition of real-time because it never blocks the mutator for more than a bounded amount of time. However, an important design issue remains to be examined. This issue is the relationship between the latency and the total memory in the system. In simple terms, a small latency means that the garbage collector will take a long time to complete a single collection cycle. During the collection cycle the mutator will create many new objects requiring the system to have a larger total memory. Conversely, a long latency means that the system needs less total memory because the collector recycles memory in a shorter cycle time. The choice of a latency time is complicated. On the one hand, the latency time must be short enough to permit the mutator (the real-time application) to meet its deadlines. On the other hand, the memory size must be balanced against cost and packaging constraints.

Figure 9 develops the relationship between latency and the amount of memory in the system. Equation 1 asserts that the maximum required size of a semi-space is less than or equal to the number of cells that are copied to, and created in, a semi-space. Equation 2 asserts that the number of cells that are copied is equal to the product of the average size of a heap segment and the number of non-garbage actors. Equation 3 asserts that the number of cells that are created during the copy is equal to the total cycle time of the garbage collector divided by the garbage collector time for each cell allocated. This equation holds because the sum of the garbage collector's time on each allocation must equal the total garbage collection cycle time. Equation 4 asserts that the total cycle time of the garbage collector is equal to the sum of the times to initialize, mark, copy and collect the garbage. Equation 5 expresses the total effort by the garbage collector in terms of actors. The amount of effort required to initialize or collect the actor is proportional to the total number of actors in the system, since each actor must be examined. As shown in [Kafura 1990], the amount of effort required to mark the actors is proportional to the square of the number of the actors in the system. The amount of effort required to copy the actors is proportional to the number of non-garbage actors in the system, since garbage heaps are not copied. Equation 6 asserts that the time per allocated cell is equal to the time per created heap segment divided by the average size of a heap segment.

Next we present an assessment of the latency by substituting the estimates in Figure 10 into Equation 7 of Figure 9. These estimates are made on the basis of our understanding of the complexity of the algorithms involved and the anticipated characteristics of actor systems. The estimates are only meant to represent and order-of-magnitude approximation.

Assume:

- M = Maximum size of a semi-space in terms of number of cells
- K = Latency (amount of effort per created heap segment)
- A = Average number of actors
- S = Average size of a heap segment in terms of number of cells
- c_1 = Constant of proportionality for effort per actor to initialize actors
- c_2 = Constant of proportionality for effort per actor to mark actors
- c_3 = Constant of proportionality for effort per actor to copy actors
- c_4 = Constant of proportionality for effort per actor to free or not free actor
- μ = Percentage of actors that are non-garbage

Then:

1. $M \leq \text{Total Cells Copied} + \text{Cells Created During Copy}$
2. $M \leq \mu AS + \text{Cells Created During Copy}$
3. $M \leq \mu AS + \text{Total GC Cycle Time} / \text{GC Time per Cell Allocated}$
4. $M \leq \mu AS + (\text{Init} + \text{Mark} + \text{Copy} + \text{Collect}) / \text{GC Time per Cell Allocated}$
5. $M \leq \mu AS + (c_1 A + c_2 A^2 + c_3 \mu AS + c_4 A) / \text{GC Time per Cell Allocated}$
6. $M \leq \mu AS + (c_1 A + c_2 A^2 + c_3 \mu AS + c_4 A) / (K/S)$
7. $M \leq AS[\mu + (c_1 + c_2 A + c_3 \mu S + c_4) / K]$

Figure 9 Latency vs. Memory Requirements

Assume:

- $c_1 = 10 \mu\text{sec}$
- $c_2 = 100 \mu\text{sec}$
- $c_3 = 10 \mu\text{sec}$
- $c_4 = 10 \mu\text{sec}$
- $S = 100 \text{ bytes}$
- 20% of the actors in the system are non-garbage*

Figure 10 Assumptions for Latency vs. Memory Evaluation

Figure 11 compares the memory requirements versus system size for varying latencies. The solid line represents the amount of memory necessary to hold the number of garbage and non-garbage actors in the system. The broken lines represent the amount of memory required by systems with the indicated latencies. The vertical distance between the solid line and a broken line represents the additional memory needed to support the real-time characteristics of the system. Figure 11 shows that for the above assumptions, a delay of 10 ms and 100 actors, an additional 20% of memory is required. For a delay of 5 ms and 1 ms, an additional 120% and 940% is required. For systems which contain 500 actors, delays of 10 ms, 5 ms and 1 ms require an additional 500%, 900%, 6,000% respectively.

Figure 12 shows the relative time that the garbage collector spends in each of its phases. The graph shows that the majority of the time is spent in the marking phase. In a system of 100 actors, the duration of the marking phase is 50 times that of the copying phase, and 1000 times that of the initialization or collection phases. For a system of 500 actors, the numbers are even more disproportionate: the duration of the marking phase is 300 times that of the marking phase, and 6000 times that of the initialization or collection phases. The duration of the time spent in the marking phase is much larger than that of other phases because it is dependent upon the square of the number of actors, whereas the other phases are linearly dependent upon the number of actors. Therefore, the best way to improve the performance of the garbage collector is to replace the marking algorithm with an algorithm that is linearly dependent on the number of actors.

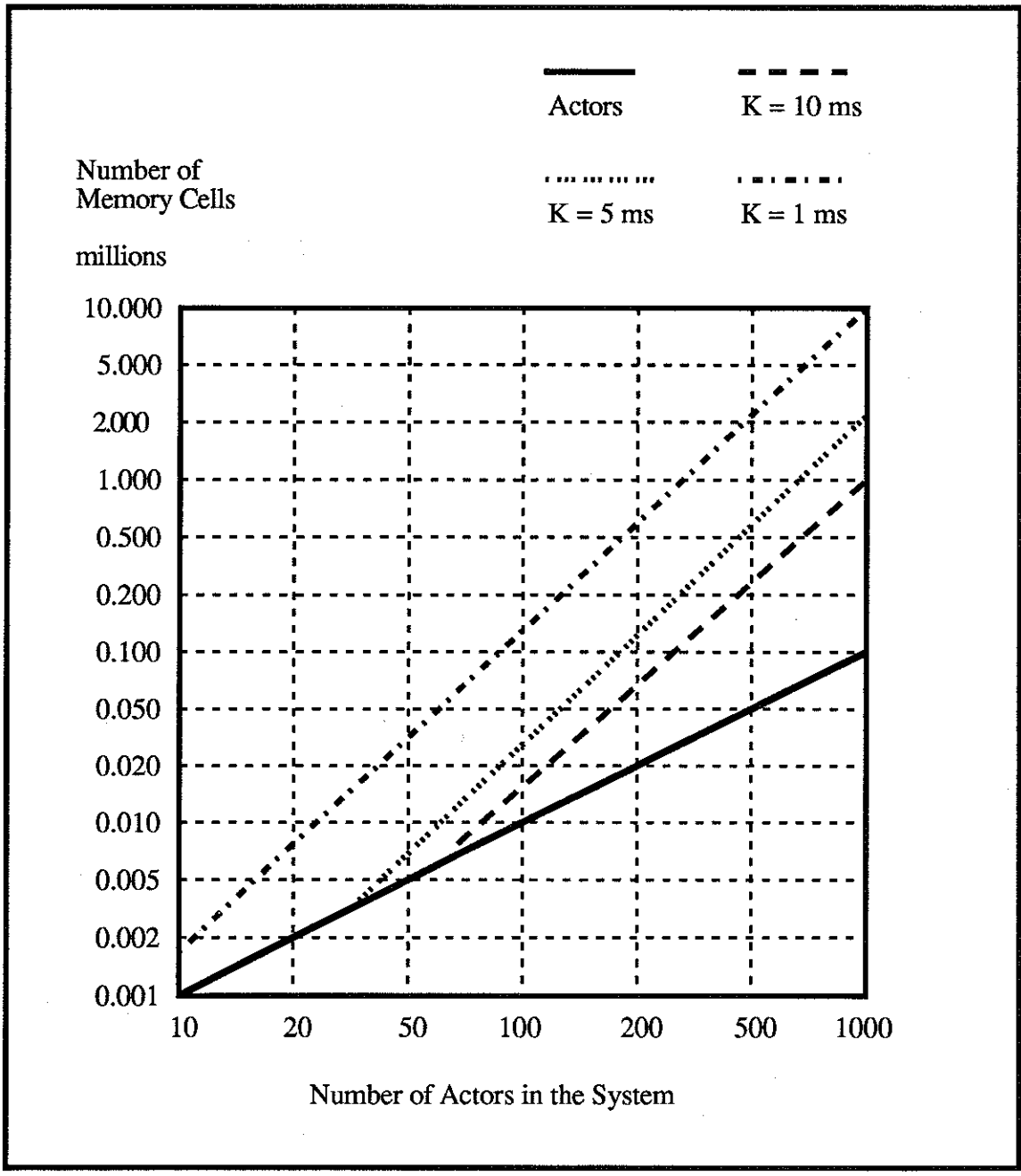


Figure 11 Comparison of Latency and Memory Requirements

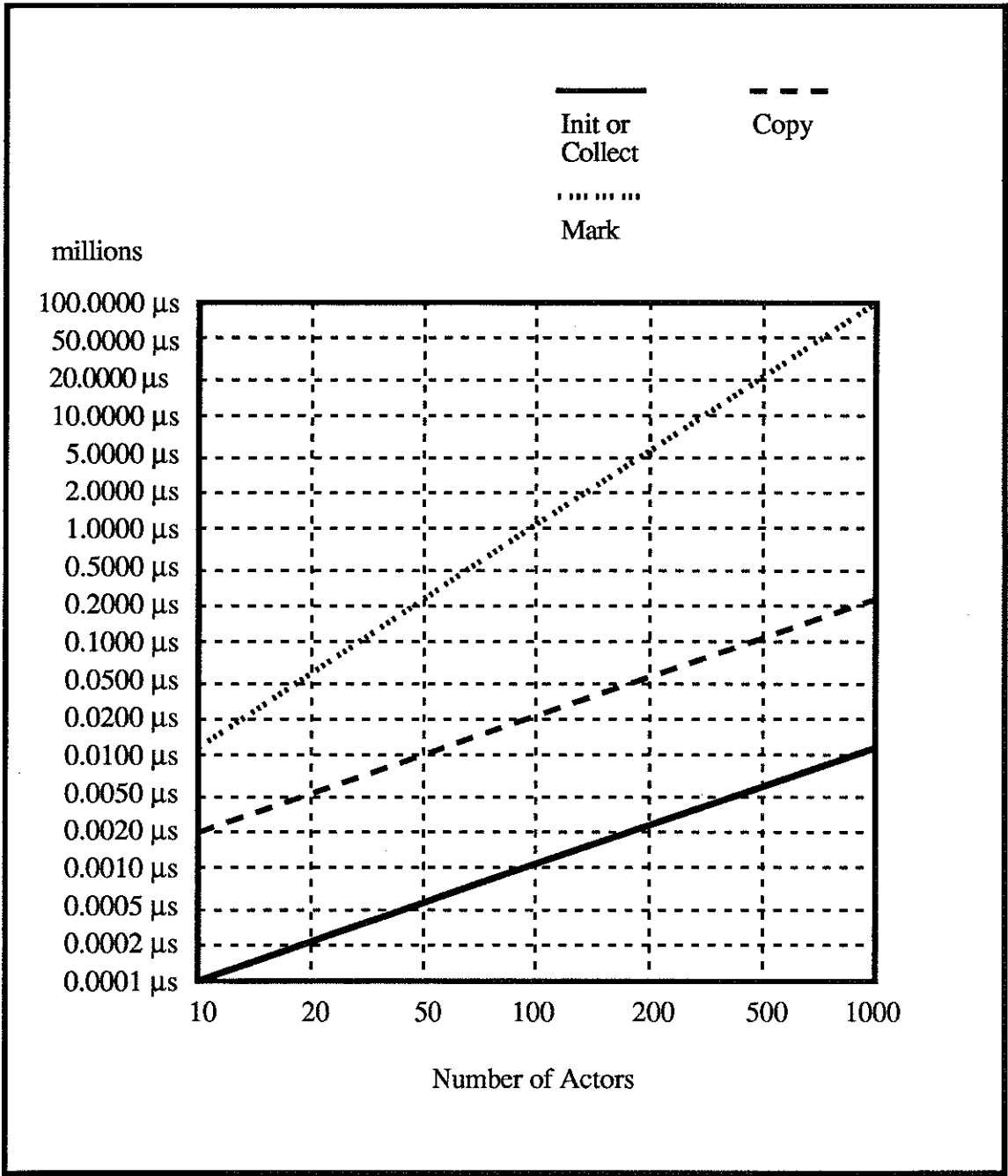


Figure 12. Relative Duration of Garbage Collection Phases

5. Conclusions and Future Directions

The analysis of latency time versus memory size presented in the preceding section clearly limits the real-time applications which can immediately exploit garbage collection and concurrent objects. Two circumstances under which these techniques can be applied are:

- low performance real-time system where an appropriate balance between memory overhead and latency can be found in the design space characterized in the preceding section, and
- multiprocessor real-time system where a processor is dedicated to the garbage collector.

The use of dedicated processors for critical system functions has been pioneered in the Spring real-time kernel [Stankovic 1987] where system scheduling is performed by a dedicated, non-application processor. Multiprocessor garbage collection performed in real-time, but for non-real-time applications, has been implemented [Appel 1988b].

The latency characteristics of the collector presented in this paper are far from desirable and also far from being optimized or sophisticated. Four approaches to improving the performance of the garbage collector are these:

- algorithm development: The algorithm developed in this paper is the first collector for actors. If the history of garbage collection algorithms is any guide, substantial performance improvements can be expected as the problem is better understood and more sophisticated algorithms are devised.

- compiler technology: The advantages of concurrent objects in real-time systems stem primarily from its logical properties (e.g., concurrency expressiveness, encapsulation) rather than its physical structure. As is the case in functional programming languages, there appear to be significant opportunities for compiler optimizations which aggregate actors so that they may be managed as a unit with reduced overhead. It is also possible in some cases to replace actor creation/collection by more efficient stack-like mechanisms thereby avoiding garbage collection entirely.
- structured topologies: The most difficult cases for the garbage collector arise in very special actor configurations (e.g., cyclic structures). There may be "structured" topologies that should be enforced - simultaneously improving the logical organization of the system and avoiding the garbage collection worst-case configurations. Tighter bounds on garbage collection performance for these structured forms can then be used.
- deadline-driven collection: Deadlines in a real-time system provides information which the garbage collector can exploit. For example, if each actor carries a time stamp indicating its deadline, then all actors whose time stamps are less than the current time may be collected. Equivalently, the time stamps may be associated with the messages.

The first two of these future direction are current research topics in our group. Progress in the last two directions must await greater experience gained through building realistic actor systems for real-time applications.

References.

- [Agha 1986] Agha, G., Actors: A Model of Concurrent Computation in Distributed Systems, M.I.T. Press, Cambridge, Massachusetts, 1986.
- [Appel 1988a] Appel, Andrew W. and Hanson, David R., "Copying Garbage Collection in the Presence of Ambiguous References," Princeton University, Department of Computer Science, Technical Report CS-TR-162-88, June 1988.
- [Appel 1988b] Appel, Andrew W., John R. Ellis and Kai Li. "Real-time Concurrent Collection on Stock Multiprocessors," Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, June, 1988.
- [Baker 1977] Baker, Henry G., Jr. and Hewitt, Carl, "The Incremental Garbage Collection of Processes," M.I.T. Artificial Intelligence Laboratory, Memo 454, Dec., 1977.
- [Baker 1978] Baker, Henry G., "List Processing in Real-time on a Serial Computer," Communications of the ACM 21(4) April 1978, pp. 280-294.
- [Black 1987] Black, Andrew, Norman Hutinson, Eric Jul, Henry Levy and Larry Carter, "Distribution and Abstract Types in Emerald," IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, p.65-76.

- [Bloom 1987] Bloom, Tony and Stanley Zdonick, "Issues in the Design of an Object-Oriented Database Programming Language," Proceedings of OOPSLA'87, October 1987, p.441-451.
- [Fenichel 1969] Fenichel, Robert R. and Jerome C. Yochelson. "A LISP Garbage-Collector for Virtual-Memory Computer Systems," Communications of the ACM 12(11) Nov., 1969, pp. 611-612.
- [Halstead 1978] Halstead, Robert Hunter, Jr. "Multiple-Processor Implementations of Message-Passing Systems," M.I.T. Laboratory for Computer Science, Technical Report 198, April, 1978.
- [Hudak 1982] Paul Hudak and Robert Keller, "Garbage Collection and Task Delection in Distributed Applicative Processing Systems," Symposium on Lisp and Functional Programming, 1982, p. 168-178.
- [Hudak 1983] Hudak, Paul, "Distributed Task and Memory Management," 2nd Annual ACM Symposium on the Principles of Distributed Computing, 1983, p. 277-289.
- [Jul 1988] Jul, Eric and Henry Levy, Norman Hutchinson, and Andrew Black, "Fine-Grained Mobility in the Emerald System," ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb., 1988, pp. 109-133.
- [Kafura 1988] Kafura, Dennis G., "Concurrent Object Oriented Real-Time Systems Research", Virginia Polytechnic Institute and State University, Department of Computer Science, Technical Report 88-47, September, 1988.

[Kafura 1990] Kafura, Dennis and Washabaugh, Doug, "Garbage Collection of Actors," Virginia Polytechnic Institute and State University, Department of Computer Science, Technical Report 90-9, March, 1990.

[Nelson 1989] Nelson, Jeff Automatic, Incremental, On-the-fly Garbage Collection of Actors, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 24061-0106, February 1989.

[Stankovic 1987] Stankovic, J.A. and K. Ramamritham, "The Design of the Spring Kernel," Proceedings of the Real-Time Systems Symposium, p. 146-157, December 1987.

[Stankovic 1988] Stankovic, John A. "Misconceptions about Real-Time Computing," IEEE Computer, October 1988, p.10-19.

[Washabaugh 1990] Doug Washabaugh, Real-Time Garbage Collection of Actors in a Distributed Systems, M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, 24061-0106, February 1990.