Non-Blocking Garbage Collection for Multiprocessors

Maurice Herlihy J. Eliot B. Moss [1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/9 September 27, 1993

Abstract

Garbage collection algorithms for shared-memory multiprocessors typically rely on some form of global synchronization to preserve consistency. Nevertheless, such global synchronization is poorly suited for asynchronous architectures: if one process is halted or delayed, other, non-faulty processes will be unable to progress. By contrast, a storage management algorithm is *non-blocking* if (in the absence of resource exhaustion) a process that is allocating or collecting memory can undergo a substantial delay without forcing other processes to block. This paper presents the first algorithm for non-blocking garbage collection in a realistic model. The algorithm assumes that processes synchronize by applying *read*, *write*, and *compare&swap* operations to shared memory. This algorithm uses no locks or busy-waiting, it does not assume that processes can observe or modify one another's local variables or registers, and it does not use inter-process interrupts.

Keywords: garbage collection; multiprocessors; shared memory; non-blocking algorithms

[1] Dept. of Comp. and Info. Sci., University of Massachusetts,Amherst, MA 01003. Eliot Moss is supported by National Science Foundation Grant CCR-8658074, by Digital Equipment Corporation, and by GTE Laboratories.

1 Introduction

Garbage collection algorithms for shared-memory multiprocessors typically rely on some form of global synchronization to preserve consistency. Shared memory architectures, however, are inherently *asynchronous*: processors' relative speeds are unpredictable, at least in the short term, because of timing uncertainties introduced by variations in instruction complexity, page faults, cache misses, and operating system activities such as preemption or swapping. Garbage collection algorithms that rely on global synchronization are poorly suited to asynchronous architectures because if one process is halted or delayed, other, non-faulty processes may also be unable to progress. By contrast, a storage management algorithm is *non-blocking* if any process can be delayed at any point without forcing all the other processes to block. This paper presents a non-blocking incremental copying garbage collection algorithm.

We note from the outset, however, that our garbage collection algorithm, like any resource management algorithm, blocks when resources are exhausted. In our algorithm, for example, a delayed process may force other processes to postpone storage reclamation, although it will not prevent them from allocating new storage. If that process has actually failed, then the non-faulty processes will eventually be forced to block when their remaining free storage is exhausted. If halting failures are a concern, then our algorithm should be combined with higher-level (and much slower) mechanisms to detect and restart failed processes, an interesting extension we do not address here. Nevertheless, our algorithm tolerates substantial delays and variations in process speeds, and may therefore be of value for real-time or "soft" real-time continuously running systems.

2 Model

There are three aspects to our model of memory: the underlying shared memory hardware and its primitive operations, the application level heap memory semantics that we will support, and the structuring of the contents of shared memory in order to support the application level semantics.

2.1 Underlying Architecture

We focus on a multiple instruction/multiple data (MIMD) architecture in which n processes, executing asynchronously, share a common memory. Each process also has some private memory (e.g., registers and stack) inaccessible to the other processes. The processes are numbered from 1 to n, and each process knows its own number, denoted by *me*. The primitive memory operations are *read*, which copies a value from shared memory to private memory, *write*, which copies a value in the other direction, and *compare&swap*, shown in Figure 1. We chose the *compare&swap* primitive for two reasons. First, it has been successfully implemented, having first appeared in the IBM System/370 architecture [11]. Second, it can be shown that some form of read-modify-write primitive is required for non-blocking solutions to many basic synchronization problems, and that *compare&swap* is as powerful in this respect as any other read-modify-write operation [7, 8]. We do *not* assume that processes can interrupt one another.

```
compare&swap(w: word, old, new: value) returns(boolean)
   if w = old
      then w := new
           return true
      else return false
   end if
end compare&swap
```

Figure 1: The Compare&Swap Operation

2.2 The Application's View

An application program has a set of private *local variables*, denoted by x, y, z, etc., and it shares a set of *objects*, denoted by A, B, C, etc., with other processes. To an application, an object appears simply as a fixed-size array of *values*, where a value is either immediate data, such as a boolean or integer, or a pointer to another object. The storage management system permits applications to create new objects, to fetch component values from objects, and to replace component values in objects. The *create* operation creates a new object of size s,[1] initializes each component value to the distinguished value *nil*, and stores a pointer to the object in a local variable.

x := create (s)

The *fetch* operation takes a pointer to an object and an index within the object, and returns the value of that component.

v := fetch (x, i)

The *store* operation takes a pointer to an object, an index, and a new value, and replaces that component with the new value.

store (x, i, v)

[1] We assume that objects do not vary in size over time, though our techniques could be extended to support such a model.

We assume that applications use these operations in a type-safe manner, and that index values always lie within range.

In the presence of concurrent access to the same object, the *fetch* and *store* operations are required to be *linearizable* [10]: although executions of concurrent operations may overlap, each operation appears to take effect instantaneously at some point between its invocation and its response. Applications are free to introduce higher-level synchronization constructs, such as semaphores or spin locks, but these are independent of our storage management algorithm.

2.3 Basic Organization

Memory is partitioned into n contiguous *regions*, one for each process. A process may access any memory location, but it allocates and garbage collects exclusively within its own region. Locations in process p's region are *local* to p, otherwise they are *remote*. Each process can determine the process in whose region an address x lies, denoted by *owner* (x). This division of labor enhances concurrency: each process can make independent decisions on when to start collecting its own region and can use its own techniques for allocation. The region structure is also well-suited for *non-uniform memory access* (NUMA) architectures (e.g., [3, 16, 18]), in which any process can reference any memory location, but the cost of accessing a particular location varies with the distance between the processor and the memory module.

An object is represented as a linked list of *versions*, where each version is a contiguous block of words contained entirely within one process's region. Versions are denoted by lower case letters a, b, c, etc. A version includes a snapshot of the vector of values of its object, and a *header* containing size information and a pointer to the next version. Version a's pointer to the next version is denoted *a.next*. A version that has a next version is called *obsolete*; a version that does not have a next version is called *current*.

An object can be referred to by pointing to any of its versions. The *find-current* procedure (Figure 2 locates an object's current version by chaining down the list of *next* pointers until it reaches a version whose *next* pointer is *nil*. The *fetch* and *store* procedures appear in Figures 3 and 4. *Fetch* simply reads the desired field from the current version. *Store* modifies the object by creating and linking in a new current version[2]. (Later, we will discuss when *store* can avoid creating new versions.) The *store* procedure is *non-blocking*: an individual process may starve if it is overtaken infinitely often, but the system as a whole cannot starve because one *compare&swap* can fail only if another succeeds. Any allocation technique can be used to implement *new*; the details are not interesting because each process allocates and garbage-collects its own region, so no inter-process synchronization is required.

[2]This method can implement arbitrary atomic updates to a single object, including read-modify-write operations, modifications encompassing multiple fields, and growing or shrinking the object size.

Multiple versions serve two purposes: first, they allow us to perform concurrent updates without mutual exclusion [9], and second, they allow our copying collector to "move" an object without locking it. In Section 5 we discuss two extensions that permit an object to be modified in place: a non-blocking technique using *compare&swap*, and a blocking technique that locks individual objects.

```
find-current(x: object) returns(object)
   while x.next ≠ nil do
         x := x.next
   end while
   return x
end find-current
```

Figure 2: Find-current: locate current version of x

```
fetch(x: object, i: integer) returns(value)
   x := find-current (x)
   return x[i]
end fetch
```

Figure 3: Fetch: obtains current contents of a slot

```
store(x: object, i: integer, v: value)
   temp := allocate local space for new version
   loop /* retry from here, if necessary */
         x := find-current (x)
         for j in 1 to x.size do temp[j] := x[j] end for
         temp[i] := v
         if compare&swap (x.next, nil, temp)
            then return
         end if
   end loop
end store
```

Figure 4: Store: Update Contents of a Slot

3 The Algorithm

Our algorithm is an incremental copying garbage collector in the style of Baker [2] as extended to multiprocessing by Halstead [12]. Each region is divided into multiple contiguous spaces: a single *to* space, zero or more *from* spaces, and zero or more *free* spaces. Initially, a process's objects reside in *from* spaces, and new objects are allocated in the *to* space. As computation proceeds, the processes cooperate to move objects from *from* spaces to *to* spaces, and to redirect reachable pointers to the *to* spaces. Once it can be guaranteed that there is no path from any local variable to a version in a particular *from* space, that space becomes *free*. When the storage allocated in a *to* space exceeds a threshold, it becomes a *from* space, and a *free* space is allocated to serve as the new *to* space. This structure is standard for copying collectors; our contribution is to show how such a collector can be implemented without forcing processes to block.

First, some terminology. A process *flips* when it turns a *to* space into a *from* space. A version residing in *from* space is *old*, otherwise it is *new*. Note that an old version may be either current or obsolete, and similarly for new versions. Further, it is possible for a new version to have an old version as its next version. Our procedures use the function *old* to test whether a version is old. This function could be implemented by associating an *old* bit with the space as a whole, or with individual objects, or by having each process maintain a table of the pages in each of its spaces.

Each process alternates between executing its application and executing a *scanning* task that checks local variables and *to* space for pointers to old versions. When such a pointer is found, the scanner locates the object's current version. If that version is old, the object is *evacuated*: a new current version is created in the scanner's own *to* space. A scan is *clean* with respect to process p if it completes without finding any pointers to versions in any of p's *from* spaces; otherwise it is *dirty*. A scan is done as follows:

1. Examine the contents of the local variables. This stage can be interleaved with assignments as long as the variables' original values are scanned before being overwritten.

2. Examine each memory location in the allocated portion of *to* space. This stage can be interleaved with allocations, as long as each newly allocated version is eventually scanned.

Scanning does not require interprocess synchronization.

How can we determine when a *from* space can be reclaimed? Define a *round* to be an interval during which each process starts and completes a scan. A *clean* round is one in which every scan is clean and no process flips. Our algorithm is based on the following claim: once a process flips, the *from* space can be reclaimed after a clean round starts and finishes.

How does one process detect that another has started and completed a scan? Call the detecting process the *owner*, and the scanning process the *scanner*. The two processes communicate through two atomic bits, called *handshake bits*, each written by one process and read by the other. Initially, both bits agree. To start a flip, the owner creates a new *to* space, marks all versions in the old *to* space as being old, and complements its own handshake bit. On each scan, the scanner reads the owner's handshake bit, performs the scan, and sets its own handshake bit to the previously read value for the owner's bit. This protocol guarantees that the handshake bits will agree again once the scanner has started and completed a scan in the interval since the owner's bit was complemented. (Similar techniques appear in a number of asynchronous shared-memory algorithms [1, 17, 15].)

How does the owner detect that all processes have started and completed a scan? The processes share an n-element boolean array *owner*, where process q uses owner$[q]$ as its "owner" handshake bit. The processes also share an n-by-n-element boolean array *scanner*, where process q uses scanner$[p][q]$ as its "scanner" handshake bit when communicating with owner process p. Initially, all bits agree. An owner q starts a round by complementing owner$[q]$. A scanner p starts a scan by copying the *owner* array into a local array. When the scan is finished, p sets each scanner$[p][q]$ to the previously saved value of owner$[q]$. The owner process q detects that the round is complete as soon as owner$[q]$ agrees with scanner$[p][q]$ for all p. An owner may not start a new round until the current round is complete.

How does a process detect whether a completed round was clean? The processes share an n-element boolean array, *dirty*. When a process flips, it sets dirty$[p]$ to *true* for all p other than itself, and when a process finds a pointer into p's *from* space, it sets dirty$[p]$ to *true*. If a process's *dirty* bit is *false* at the end of a round, then the round was clean, and it reclaims its *from* spaces. The process clears its own *dirty* bit before starting each round.

We are now ready to discuss the algorithm in more detail. To flip (Figure 5), a process allocates a new *to* space, marks the versions in the old *to* space as old, sets everyone else's *dirty* bit, and complements its *owner* bit. (A process may not flip in the middle of a scan.) To start a scan (Figure 6), the process simply copies the current value of the *owner* array into a local array. The scanner checks each memory location for pointers to old versions (Figure 7). When such a pointer is found, it sets the owner's *dirty* bit, and redirects the pointer to a new current version, evacuating the object to its own *to* space if the current version is old. When the scan completes (Figure 8), the scanner informs the other processes by updating its *scanner* bits to the previously-saved values of the *owner* array. The scanner then checks whether a round has completed. If the round is completed and its dirty bit is *false*, the process reclaims its *from* spaces. If the round is completed but the dirty bit is *true*, then the process simply resets its *dirty* bit. Either way, it then starts a new scan.

```
flip()
    mark versions in current to space as old
    create new to space
    for i in 1 to n do
        if i ≠ me
            then dirty[i] := true
        end if
    end for
    owner[me] := not owner[me]
end flip
```

Figure 5: Starting a flip

```
scan-start()
    for i in 1 to n do
        local-owner[i][me] := owner[i]
    end for
end scan-start
```

Figure 6: Starting a scan

4 Correctness

For our algorithm there are two correctness properties of interest: *safety*, ensuring that the algorithm implements the application-level model described in Section 2.2, and *liveness*, ensuring that as long as processes continue to take steps, then garbage is eventually collected. We now discuss each in turn.

4.1 Safety

There are two safety properties to be demonstrated: that the implementations of the model's basic operations are linearizable, and that non-garbage objects are never collected.

4.1.1 Linearizability of the Basic Operations

One way to show an operation implementation is linearizable is to identify a single primitive step where the operation "takes effect" [14]. For *fetch*, this instant occurs when it reads a null *next* pointer, and for *store*, when its *compare&swap* succeeds in replacing a null *next* pointer with a pointer to its new version. Note that *scan* is essentially a *store* that does not affect the logical contents of the object.

```
scan-value(x: object) returns(object)
   if old (x) then dirty[owner (x)] := true end if
   loop /* evacuate object if necessary */
         x := find-current (x)
         if new (x) then return x end if
         temp := allocate local space for new version
         for j in 1 to x.size do temp[j] := x[j] end for
         if compare&swap (x.next, nil, temp)
            then return temp
            else  release local space for new version
         end if
   end loop
end scan-value
```

Figure 7: Scanning a pointer

4.1.2 Only Garbage is Collected

Claim 1 *Every process starts and completes at least one scan during the interval between the start and end of* p*'s clean round.*

Proof: Since p reset owner[p] to disagree with each scanner[q][p] at the start of the interval, and since these values agree again at the end, each process q must have (1) read the new value of owner[p], (2) performed a scan, and (3) set scanner[q][p] to the value of owner[p]. ∎

Claim 2 *Every process starts and completes at least one scan clean with respect to* p *during the interval between the start and end of* p*'s clean round.*

Proof: Each process completed a scan (Claim 1) but no process set p's dirty bit. ∎

Claim 3 *When a process reclaims a* from *space, no path exists into that space from any other process's local variables.*

Proof: Suppose otherwise: p completes a clean round even though some process has a path from a local variable to a version x in p's *from* space. If such a path exists at the end of the clean round, then some path must have existed at the start of the round. Call such a path an *early* path.

Suppose some early path passes through a new version. Let y be the last new version on the path from the variable to x. Because the round is clean, no process flips, and y remains new for the duration of the round. The scanning

```
scan-end()
    /* Notify other from spaces */
    for i in 1 to n do
        scanner[i][me] := local-owner[i]
    end for
    /* Did a round complete? */
    if (∀i) scanner[i][me] = owner[me] then
        if not dirty[me]
            then reclaim from spaces
        end if
        dirty[me] := false
    end if
    /* start new scan */
    scan-start()
end scan-end
```

Figure 8: Completing a scan

process will eventually inspect y, and it will evacuate the old versions referenced by y, the old versions they reference, and so on. When the scanning process reaches x, it sets dirty[p], contradicting the hypothesis that the round was clean.

If no early path passes through a new version, then some process q has a local variable holding a pointer that references x through a chain of old versions. Any such local variable must be overwritten before q starts its clean scan, since otherwise q would scan the variable, start evacuating old versions, and set dirty[p] when it reaches x. If all such local variables are overwritten without being stored, then there would be no path to x at the end of the round. Therefore, some local variable v must have been stored in a new version y after the start of the clean round, but before v was overwritten, and before the start of q's clean scan. By the argument given in the previous paragraph, q's next scan inspects y, evacuates x, and sets dirty[p], again contradicting the hypothesis that the round was clean. ∎

4.2 Liveness

We claim that if each process always eventually scans, then some process always eventually reclaims its *from* spaces. Suppose not. Each process will eventually exhaust its finite supply of free spaces, further flips will cease, and *dirty* bits will be set only by the scanner. Since each process continues to scan, each process observes an infinite sequence of rounds, where each round includes a dirty scan. Each dirty scan, however, reduces the number of reachable objects whose current versions are old, since each object reachable from a *to* space or from local variables is evacuated. Since the supply of objects is finite, all objects will eventually have new current versions. In the next round, all pointers are redirected to current versions, and in the round after that, all scans are clean, a contradiction.

Finally, any process that always eventually flips will eventually have no versions of unreachable objects in *to* space. When a process creates a new *to* space, it evacuates only those objects reachable from its local variables at the time of the flip, or objects created after the flip. Therefore, once an object becomes unreachable, it will have no versions in *to* space once each process does a flip.

5 Extensions

We now describe four interesting extensions to our algorithm. The first two allow objects to be updated without creating new versions. The third extension allows some *from* spaces to be reclaimed sooner. Finally, we consider making our copy collection scheme generational.

5.1 Non-Blocking Update in Place

Creating new versions of an object may be expensive if objects are large or modifications are frequent. In this section, we show how to reduce this cost by permitting a process to make in-place modifications to versions in its own *to* space. We add the following fields to the version header: *a.seq* is a modulo two sequence number for the next update, initially distinct from the value in the *next* field, *a.index* is the index of the slot being updated, and *a.value* is the new value for that slot. The type of the *next* field is extended so that it may hold either a pointer to the next version or a sequence number. There need be only two values for sequence numbers: if $a.seq = a.next$, then the current update is installed, and otherwise it is ignored.

To perform a *store*, a process chains down the list of versions until it finds a version whose *next* field is either *nil* or a sequence number. If the version is remote, the *store* proceeds as before. If the version is local, however, the process calls the *local-store* operation shown in Figure 9. The operation takes a pointer to the version, the value observed in the *next* field, the index of the slot to be modified, and the new value of the slot. The process calls *compare&swap* to reset *a.next* from its current value (either a sequence number or *nil*) to the new sequence number. If it succeeds, the process scans the old value and updates the target slot. (It is necessary to scan the overwritten value to preserve the invariant that the scan inspects every value written to *to* space.) If it fails, the process locates the newer version and starts over. The restriction that update in place be performed only by the owning process is well-suited to a NUMA architecture, where it is more efficient to update closer objects.

When a remote process attempts to update a version, it creates a local copy just as before. One extra step is needed: after copying the version, it checks whether *x.next* is equal to *x.seq*. If so, the storing process must complete the pending updates by scanning slot *x.index* and storing *x.value* in that slot. The evacuate procedure is similarly affected. (These changes are not shown.) The *fetch* operation need not be modified, because observing the *next* field linearizes every *fetch* with respect to operations that create new versions, and observing the updated field linearizes the *fetch* with respect to updates in place.

```
store-local(x: object, next: value, i: integer, v: value)
   seq := next + 1 (mod 2)
   x.seq := seq /* it is important to set this first */
   x.index := i
   x.value := v
   if compare&swap (x.next, next, seq)
      then scan(x[i])
           x[i] := v
      else  store (x, i, v)
      end if
   end store-local
```

Figure 9: Store, with update in place

5.2 Blocking Update in Place

Perhaps the most practical approach to performing updates in place is to relax slightly our prohibition on mutual exclusion by allowing the current version's owner to lock out concurrent accesses. The principal advantage of this approach is that updates do less work, especially if the application is going to lock the object anyway, or if the likelihood of conflict is low. The disadvantage, of course, is that the storage management algorithm now permits one process to force another to block. Nevertheless, even if storage management is no longer non-blocking, allocation and garbage collection are still accomplished without global synchronization.

As before, only the owner of the current version may update an object in place. The owner locks an object as follows: (1) it calls *compare&swap* to set the current version's *next* field to a distinguished *locked* value, (2) it scans the current values of the fields that will be updated, (3) it operates on the object, (4) it rescans the updated fields, and (5) it unlocks the object by setting the *next* field back to *nil*. *Fetch* and *Store* are changed so that a process that encounters a locked version waits until the *next* field is reset to *nil*.

Since the owner is the only process that updates the object in place, there is no need to synchronize with the scanner, except perhaps to avoid superfluous scans. Step (2) insures that values possibly seen by other processes will be scanned, similar to the scan in *store-local.* Step (4) insures that if the object has already been scanned, the new values will not be mistakenly omitted. Depending on the details of the incremental scanning process, it is correct to omit step (2) or step (4) on some occasions.

5.3 Reclaiming *From* Spaces Earlier

Rather than reclaiming each process's *from* spaces all at once, we can reclaim them individually, by keeping more detailed information about pointers encountered while scanning. Rather than associating *dirty* bits with each process, we associate them with each *from* space. When a scanning process encounters a pointer into *from* space s, it sets the *dirty* bit for space s. At the end of a scan, each *from* space whose *dirty* bit is *false* can be reclaimed. If a space's *dirty* bit is *true*, then the *dirty* bit is cleared and a new scan is started. When a flip occurs, the *dirty* bits of *all* other processes' *from* spaces must be set.

5.4 Generational collection

Extending our algorithm to generational incremental collection is straightforward. We divide each process's region into some number of generations, ordered by age. Pointers from older to younger generations are kept in *remembered* sets, reducing the work necessary to scan older generations. It seems sensible also to remember pointers to remote objects, to further reduce the need to scan objects. Additionally, some means must be provided for a process to discover old versions in old generations without scanning the old generations. One way to do that is to have a bit table, with one bit per some fixed number of words (Wilson calls this *card marking* [20]). When a new version is installed, the process that created the new version sets the bit corresponding to the address of the previously current version. The owner of that version can then locate the old version by scanning the bit table and the associated memory words rather than scanning all memory in the old generations. The partitioning of regions into generations is an internal concern of the processes, although care must be taken that the region is scanned correctly.

6 Related Work

Our algorithm is an intellectual descendant of Baker's single-processor algorithm [2], and can be viewed as a non-blocking refinement of Halstead's multiprocessor algorithm [12]. Our algorithm differs from Halstead's because it does not require processes to synchronize when flipping *from* and *to* spaces, and we do not require locks on individual objects.

A number of researchers [4, 5, 13] have proposed two-process mark-sweep schemes, in which one process, the *mutator*, executes an arbitrary computation, while a second process, the *collector*, concurrently detects and reclaims inaccessible storage. The models underlying these algorithms differ from ours in an important respect: they require that the collector process observe the mutator's local variables, which are treated as roots. Many current multiprocessor architectures, however, cannot meet this requirement, since the only way to copy a pointer is to load it into a private register, and then store it back to memory, leaving a "window" during which the collector cannot tell which objects are referenced by the mutator. These algorithms synchronize largely through read and write operations, although some kind of mutual exclusion appears to be necessary for the free list and other auxiliary data structures. Pixley [19] gives a generalization of Ben-Ari's algorithm in which a single collector process cleans up after multiple concurrent mutators. This algorithm, as Pixley notes, behaves incorrectly in the presence of certain race conditions, which Pixley explicitly assumes do not occur. Our algorithm introduces multiple versions to avoid precisely these kinds of problems.

The Ellis, Li, and Appel [6] describe the design and implementation of a multi-mutator, single-collector copying garbage collector. This algorithm is blocking, since processes synchronize via locks, and flipping the *from* and *to* spaces requires halting the mutators and inspecting and altering their registers.

7 Conclusions

The garbage collection algorithm presented here is (to our knowledge) the first shared-memory multiprocessor algorithm that does not require some form of global synchronization. The algorithm's key innovation is the use of asynchronous "handshake bits" to detect when it is safe to reclaim a space. There are several directions in which this research could be pursued. First, as noted above, although our algorithm tolerates delays, it does not tolerate halting failures, since *from* space reclamation requires a clean sweep from each process. It would be of great interest to know whether halting failures can be tolerated in this model, and how expensive it would be. Second, our algorithm make frequent copies of objects. Some copying, such as moving an object from *from* space to *to* space, is inherent to any copying collector. Other copying, such as moving an object from one process's *to* space to another's, is primarily intended to avoid blocking synchronization, although it might also improve memory access time in a NUMA architecture. The "pure" algorithm also copies objects within the same *to* space, although this copying can be eliminated either by adding extra fields (non-blocking) or by locking individual objects (blocking). It would be useful to have a more systematic understanding of the trade-offs between copying and blocking synchronization. Finally, it would be instructive to gain some practical experience with this (or similar) non-blocking algorithms.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. In *Ninth ACM Symposium on Principles of Distributed Computing*, 1990.

[2] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[3] BBN. The uniform system approach to programming the Butterfly parallel processor. Technical Report 6149, Bolt, Beranek, and Newman Adv. Computers, Inc., Cambridge, MA, October 1985.

[4] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffins. On-the-fly garbage collection: An excercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[6] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, February 1988.

[7] Maurice P. Herlihy. Wait-free synchronization. To appear, ACM TOPLAS.

[8] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[9] Maurice P. Herlihy. A methodology for implementing highly concurrent data structures. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizabilty: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] IBM. System/370 Principles of Operation. Order Number GA22-7000.

[12] R.H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[13] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *18th Symposium on Foundations of Computer Science*, pages 120–131, October 1977.

[14] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[15] Leslie Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[16] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, New Haven CT, September 1986.

[17] G. L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[18] Greg H. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *International Conference on Parallel Processing*, 1985.

[19] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–49, December 1988.

[20] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *OOPSLA '89 Conference Proceedings*, volume 24(10) of *ACM SIGPLAN Notices*, pages 23–35, New Orleans, LA, October 1989. ACM.