Lock-Free Garbage Collection for Multiprocessors

Maurice P. Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge, MA 02139
herlihy@crl.dec.com

J. Eliot B. Moss*
Dept. of Comp. and Info. Sci.
University of Massachusetts
Amherst, MA 01003
moss@cs.umass.edu

Abstract

Garbage collection algorithms for shared-memory multiprocessors typically rely on some form of global synchronization to preserve consistency. Such global synchronization may lead to problems on 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 *lock-free* if (in the absence of resource exhaustion) a process that is allocating or collecting memory can be delayed at any point without forcing other processes to block. This paper presents the first algorithm for lock-free 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, busy-waiting, or barrier synchronization, 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.

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 may lead to undesirable blocking on 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 *lock-free* if any process can be delayed at any point without forcing any other process to block.[1] This is a very strong view of blocking, since even very short term locks could lead to blocking in our sense. The benefit of this view is that we can make a strong guarantee of progress if a system is lock-free. This paper presents a lock-free 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 any free storage is available. 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.

Previous algorithms typically include two distinct forms of synchronization. One is synchronization of access, update, etc., to individual objects, which we call *local synchronization*. For example, Halstead [Hal-

*Eliot Moss is supported by National Science Foundation Grant CCR-8658074, by Digital Equipment Corporation, Apple Computer, and GTE Laboratories.

[1] Note that we do not use *blocking* to mean that ordinary execution, e.g., *mutator* processes, stops during collection. In fact, we do not make a mutator/collector distinction, as will be seen.

stead, 1985] uses short term locks on objects. The other is some form of barrier for phases of the garbage collection computation and/or locks on such data structures as a free list. This we call *global synchronization*. Our algorithm is lock-free in both local and global synchronization, and distinct techniques are used for each.

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 do *not* assume that processes can interrupt one another.

We chose the *compare&swap* primitive for two reasons. First, it has been successfully implemented, having first appeared in the IBM System/370 architecture [IBM]. 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 [Herlihy, 1991; Herlihy, 1988]. Most multiprocessors, even ones based on load/store architectures, have primitives of adequate power. For example, the forthcoming MIPS II architecture [Kilian, 1991] includes two relevant instructions, Load Linked and Store Conditionally. The first does an ordinary load but sets a special status bit in the processor called the LL bit. This bit is automatically cleared if an underlying cache consistency protocol detects updates that might affect the location previously loaded. The Store Conditionally instruction, which is required to store into the location previously loaded, performs the store only if the LL bit is still set, and returns the LL bit value. It is easy to implement any conditional or unconditional, single memory location, read-modify-write operation with these two instructions, including *compare&swap*.

Note that we assume that *compare&swap* forces appropriate cache consistency, not only for the location updated, but also for most previous writes (certainly writes to the same object, and possibly other writes to shared memory as well). It is easy to examine our code sequences and determine the exact cache consistency requirements, so we omit the details.

```
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,[2] 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)

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

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

In the presence of concurrent access to the same object, the *fetch* and *store* operations are required to be *linearizable* [Herlihy and Wing, 1990]: 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., [BBN, 1985; Li, 1986; Pfister *et al.*, 1985]), 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[3]. Later we will discuss how *store* can avoid creating new versions. The *store* procedure is *lock-free*: 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.

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

Any allocation technique can be used to implement *create*; the details are not interesting because each process allocates and garbage-collects its own region, so no inter-process synchronization is required.

Multiple versions serve two purposes: first, they allow us to perform concurrent updates without mutual exclusion [Herlihy, 1990], and second, they allow our copying collector to "move" an object without locking it. In Section 5 we discuss approaches to performing updates in place.

```
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 := 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: updates contents of a slot

3 The Algorithm

Our algorithm is an incremental copying garbage collector in the style of Baker [Baker, 1978] as extended to multiprocessing by Halstead [Halstead, 1985]. 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 any version in a particular *from* space, that *from* 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 a lock-free implementation of such a collector.

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 maintaining a table mapping memory pages to 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, without locks or barrier synchronization, 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 [Afek *et al.*, 1990; Peterson, 1983; Lamport, 1986].)

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 sets its own *dirty* bit to *false* 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 the

scanner's dirty bit is *false*, the scanner reclaims its *from* spaces. If the round is completed but the dirty bit is *true*, then the scanner 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

```
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 := 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 space for new version
       end if
  end loop
end scan-value
```

Figure 7: Scanning a pointer

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 outline the arguments here; more details may be found in [Herlihy and Moss, 1992].

```
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

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. One way to show an operation implementation is linearizable is to identify a single primitive step where the operation "takes effect" [Lamport, 1983]; in this case the last access to the *next* field of an object is such a primitive step.

The argument that only garbage is collected proceeds by demonstrating these three claims:

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

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.*

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

Liveness is shown by proving this claim:

Claim 4 *If each process always eventually scans, then some process always eventually reclaims its* from *spaces.*

5 Update in place

A significant obstacle to general practical use of our algorithm is the requirement to create a new version for each update. However, inspired by [Massalin and Pu, 1991], we devised a very simple technique for update in place using the *cas-two* operator, defined in

Figure 9. Later versions of the M68000 architecture define a CAS2 instruction that implements this operator [Motorola, Inc., 1989], so our algorithm is practical, at least on that architecture. The *cas-two* operator may be difficult to incorporate smoothly into RISC architectures; for example, the previously mentioned MIPS II instructions are inadequate for implementing *cas-two* directly.

```
compare&swap-two
    (w1, w2: word, o1, o2, n1, n2: value)
  returns(boolean)
  if w1 = o1 and w2 = o2
    then w1 := n1
         w2 := n2
         return true
    else return false
  end if
end compare&swap-two
```

Figure 9: The Compare&Swap-Two operation

In using *cas-two* for update in place the idea is to verify that the *next* pointer is still nil *and* to do the update in the *same* atomic step. Figure 10 shows the revised *store* routine. Note that versions are still needed for garbage collection, and are permitted, but no longer required, for *store* operations. Making new versions might be sensible, e.g., to increase locality on a NUMA multiprocessor.

```
store-cas-two(x: object, i: integer, v: value)
  loop /* retry from here, if necessary */
       x := find-current (x)
       if cas-two (x.next, x[i], nil, x[i], nil, v) then
         return
       end if
  end loop
end store-cas-two
```

Figure 10: Update in place using *cas-two*

Unfortunately, few architectures now include *cas-two*. It is possible to allow a single writer, namely the owner, to update in place by adding extra fields to each object and requiring creators of new versions to examine the additional fields for a pending update. Another approach is to abandon lock-free implementation of local (per-object) synchronization, and to use short term locks. Details are available in an expanded version of the paper [Herlihy and Moss, 1992], which discusses some additional extensions as well.

6 Related Work

Our algorithm is an intellectual descendant of Baker's single-processor algorithm [Baker, 1978], and can be viewed as a lock-free refinement of Halstead's multiprocessor algorithm [Halstead, 1985]. 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 [Ben-Ari, 1984; Dijkstra *et al.*, 1978; Kung and Song, 1977] 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. The problem is that one processor generally cannot examine another processor's registers, and the registers are a crucial part of the state of 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 [Pixley, 1988] 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 [Ellis *et al.*, 1988] 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.

Massalin and Pu [Massalin and Pu, 1991] describe how to implement an operating system kernel without locks. From them we realized the existence and usefulness of the *cas-two* operator; they also appear to have introduced the term *lock-free*. Beyond that there is little similarity between our work and theirs since they were considering lock-free solutions to different problems.

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 innovations are lock-free object operations for local synchronization and the use of asynchronous "handshake bits" for global synchronization, 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 makes 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 by using a stronger operator (*cas-two*), by adding extra fields (owner-only update in place), or by locking individual objects. It would be useful to have a more systematic understanding of the trade-offs between copying, blocking synchronization, and the power of synchronization operators. Third, it appears that *cas-two* allows substantially more efficient implementation of our algorithms and it would be helpful to have a precise formal characterization and proof of this conjecture. Fourth, since our algorithms assume enough resources are available to prevent blocking resulting from resource exhaustion, it would be helpful to have a quantitative analysis of the resources required to prevent exhaustion, and a qualitative development of reasonable assumptions leading to practical guarantees that resources will not be exhausted. Finally, it would be instructive to gain some practical experience with this (or similar) lock-free algorithms. The version of the algorithm that uses *cas-two* appears to be practical; other versions may be practical in more limited circumstances, e.g., when objects are updated infrequently.

References

[Afek *et al.*, 1990] 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).

[Baker, 1978] Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[BBN, 1985] BBN. The uniform system approach to programming the Butterfly parallel processor. Tech. Rep. 6149, Bolt, Beranek, and Newman Adv. Computers, Inc., Cambridge, MA, Oct. 1985.

[Ben-Ari, 1984] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (July 1984), 333–344.

[Dijkstra *et al.*, 1978] 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. *Commun. ACM 21*, 11 (Nov. 1978), 966–975.

[Ellis *et al.*, 1988] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Tech. Rep. 25, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, Feb. 1988.

[Halstead, 1985] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

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

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

[Herlihy, 1991] Maurice P. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (Jan. 1991), 124–149.

[Herlihy and Moss, 1992] Maurice P. Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* (1992). To appear.

[Herlihy and Wing, 1990] Maurice P. Herlihy and Jeannette M. Wing. Linearizabilty: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

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

[Kilian, 1991] Earl Kilian, Apr. 1991. Personal communication.

[Kung and Song, 1977] 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* (Oct. 1977), pp. 120–131.

[Lamport, 1983] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 190–222.

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

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

[Massalin and Pu, 1991] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, Department of Computer Science, New York, NY, Mar. 1991.

[Motorola, Inc., 1989] Motorola, Inc. *M68000 Family Programmer's Reference Manual*. Motorola, Phoenix AZ., 1989. Document M68000PM/AD.

[Peterson, 1983] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 46–55.

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

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