Concurrent, Asynchronous Garbage Collection Among Cooperating Processors

Charles R Lang

Computer Science Department
California Institute of Technology

4724:TR:82

Concurrent, Asynchronous Garbage Collection Among Cooperating Processors

Charles R. Lang

California Institute of Technology
Computer Science Department
Pasadena, California 91125

Technical Report #4724

February 8, 1982

The work described in this document was sponsored by the Defense Advanced Research Projects Agency, ARPA order 3771, and monitored by the Office of Naval Research under contract N00014-79-C-0597.

1. Introduction

Presented here is a scheme for the identification and elimination of unreachable program objects in a large multi-processor environment. The problem of garbage collection has been an interesting problem for many years among the implementors of various languages such as Algol 68 [Van Wijngaarden69], Simula 67 [Birtwhistle73] and Lisp [McCarthy60] which provide for dynamic allocation of data structures. Garbage collection has been part of operating systems to a lesser degree for sometime but took on a new importance with the implementation of Hydra on C.mmp [Wulf72,80]. Here, the operating system is distributed over a number of processors and therefore, collection of garbage must take place across a number of distinct address spaces concurrently with the operation of a number of processors [Almes80].

Various implementations of Lisp have dealt with the problem of garbage collection. Early work [McCarthy60,Collins60] provided garbage collection for Lisp on a single processor. More recently, considerable effort has been given to the use of multiple processors to execute Lisp with at least one of them responsible for garbage collection [Steele75,Deutsch76,Wadler76]. The algorithm proposed by Dijkstra, et al [Dijkstra78] has been proved correct [Gries77]. In all cases, these algorithms are presumed by their authors to operate in a system where every processor has equal or near equal access to a single address space. Moreover, the problem of garbage collection in Lisp is somewhat more restricted than the more general case of Simula [Arnborg72], insofar as Lisp objects are of fixed size and Lisp data structures may be of a restricted topology.

Garbage objects can be identified in systems by reference counting [Collins60]. This technique can be applied to most systems, even those with more than one processor. However, reference counting suffers from two problems for which no acceptable solutions come to mind. First, self-referential data structures or data structures with cyclic graphs can not be identified as garbage by this method. For some environments, those which restrict the user to tree-like structures, this problem may be tolerable but in a more general system it is not. Reference counting also involves a very large computational overhead to keep the reference counts up to date. Since each object's reference counter must be modified whenever and wherever a pointer to that object is copied or overwritten, many simple operations become complex. In a multiprocessor system, this overhead is manifested either by a high communication traffic or by a large number of memory accesses used to update reference counts as pointers are manipulated.

If, as Backus [Backus78] suggests, programming can be liberated from the von Neumann style, the hardware structures that are constructed to support these new styles must certainly avoid the von Neumann bottleneck. This bottleneck is the narrow pipe through which all memory accesses must flow in conventional machines. This bottleneck is present, and is even more choked, in machines with multiple processors connected to a single memory. The choking may be somewhat relieved by increasing the cost and complexity of the system with such techniques as the interleaving of memory, crossbar switches and other "stunt" boxes [Thornton70], but the effectiveness of these techniques is necessarily limited by space, time and cost considerations.

If the von Neumann bottleneck is to be removed, then its large, monolithic address space must become distributed among the various processors of a system. Also, the semantics of an "address" will grow beyond its current meaning which defines it as a fixed size word in a very large set of words. In this paper, the address becomes a reference variable or pointer referring to an object. Objects are the basis units from which data structures are built. Procedural

attributes are defined for classes of objects and become a set of operations that manipulate the data contained in an instance of an object. These concepts are found in Simula and other languages and have been extended to operating systems by Hydra [Jones77]. In a system consisting of numerous processors each with their own "object memory", all access to an object within the memory of a particular processor is controlled by that processor. Objects communicate by passing messages to other objects for which they contain a reference variable.

Such a system provides some simplifications not possible in the Lisp systems but also introduces new complexities. If each processor is to control the access and function of its own set of objects, then the need for notions of mutual exclusion, critical regions and indivisibility in the operations of the processor is eliminated since it is the only entity in contact with its objects. However, the iterations among the processors of the system raise new problems, such as their synchronization if they are to perform such tasks as garbage collection.

2. The Object-Oriented Environment

The environment under consideration here is one containing a large number of processors, from as few as 16 to perhaps 64K (K=1024). Some interconnection is presumed to exist by which the processors communicate with each other in the course of their normal operation. No presumption is made about the topology or bandwidth of the connections. However, it is clear that the communication capabilities of the network must be high to support the execution of interesting problems. Each processor contains its own memory to which it has exclusive access. Each processor/memory node runs its own operating system, which may be better thought of as a run-time system. This code is always resident in the processor's memory and serves to support a particular programming environment that pervades the entire system of processors. The identical code, running in each processor, implements an object-oriented machine.

Objects, in this environment, are similar in their nature to the objects found in Simula and Smalltalk [Ingalls78]. They are each separate processes which communicate with other objects via a message passing facility. All objects of a given "class" share the same code but each instance of an object has its own state. This state may consist of the usual real, integer, boolean and text data types common to most programming languages as well as arrays of these same types. More importantly, objects may contain pointers to other objects. These pointers merely address the indicated object and identify its type, they do not indicate the physical location of the object. This type of pointer will be called a reference variable using Simula nomenclature. Any number of reference variables may refer to a given object.

Objects can have a broader interpretation in existing systems. The files and directories in the file system of a machine running UNIX[1] [Ritchie74] could be thought of as objects of whose type is implied by the operating system. If a number of such machines are connected via a network and links are permitted in a machine pointing at files or directories in another machine, essentially the same object-oriented situation can be seen to exist. Distributed database systems [Yu81] permitting multiple, concurrent access have a similar need to resolve the status of objects residing in multiple machines.

---

[1] UNIX is a Trademark of Bell Laboratories.

Objects are distributed among the machines and may be moved between them at any time. Objects are constrained to fit wholely within any given processor. They may execute concurrently where provided for by the programmer and where the opportunity exists. Objects may create other objects but cannot explicitly destroy other objects. Reference variables may be overwritten, copied and sent to other objects in messages. These operations result in a dynamic system where both the positions of the objects and the topology of their pointers change continuously. There can be no restrictions on the kinds of structures that might be generated (e.g. cycles in the pointers must be permitted). There is no way to enforce such restrictions nor is there any desire to.

The memory associated with each processor may include disk storage. If mass storage peripherals are present on a processor, they can be regarded as logically part of the processor's memory. No distinction is drawn in this paper between objects stored in random access memory and those stored on disks if they are under the control of the same processor.

The run-time system, resident in each processor, provides for the passing of messages between objects. This mechanism is transparent and hides the actual physical location of the objects from other objects. The run-time system may also cause objects to be moved about between processors and it multiplexes its processor among the executable objects in its memory. The object of this paper is the portion of the run-time system that implements the collection of garbage objects.

Briefly, the environment consists of a large number of objects with a structure of pointers between them of an arbitrary topology. The objects are distributed over a number of processors and those objects that are executing may change the topology of the pointers. The object of garbage collection is to identify and eliminate those objects which are unreferenced in the system and are idle.

There are two reasons for eliminating such objects. First, while the "object space" is virtual, real space must exist on a device such as a disk. It is probable that such a system would have a number of disk drives distributed among the processors for the purpose of providing an "object swapping" facility or a long-term storage for files. Since any such devices are finite, the space occupied on them by garbage objects must be recovered to be used by newer, non-garbage objects. Second, the unique identifiers associated with each object are also a limited resource and those assigned to objects that become garbage must be recovered for reuse. When an object is collected as garbage, its identifying name must be made available to that part of the run-time system that generates new objects.

3. A Description of the Algorithm

This paper presents an algorithm that collects the garbage in the environment described above. Every processor node which executes, stores or otherwise manipulates objects must run a task in the backround which is part of the overall garbage collection process. These tasks, each in one of the processors in the network, communicate with a central entity which maintains overall control of each phase of the garbage collection. The communication between the central process and the various tasks in the processors is of a very low bandwidth and serves only to synchronize the other processors in a very coarse way with respect to the stages of the garbage collection. The controlling entity may be a

separate processor with dedicated communication facilities to connect it to the other processors, or it may be merely another backround task executing in any processor and communicating via the same medium as the objects in the system.

There are three phases to the collection process. The first is to clear any and all mark bits in the entire system. The second is to mark those objects which are not garbage. And third, all unmarked objects are collected and the resources they occupy are made available for allocation to new objects.

For the sake of clarity, some definitions are needed:

(1) An executable object is one which has one or more messages to process or is actually processing a message. Executable objects are defined to be non-garbage for as long as they are executable.

(2) There may be a "root" object which is known to be the progenitor of all other objects. If there is such an object, this object is defined to be non-garbage.

(3) A path is a series of reference variables which point from one object to another, where an object pointed to contains the next pointer of the path.

(4) Any object for which there exists at least one path from any executable object to itself is not garbage. Conversely, any object for which no path exists from an executable object is garbage.

This definition of garbage differs from more traditional definitions only in that an object, or set of objects, isolated from the rest of the system is not garbage if at least one of the set can be executed. This aspect of the definition will be seen to provide a large increase in the efficiency of the algorithm.

The following conditions must also apply:

(1) Each object must have an attribute of MARKED. This attribute is TRUE after the processor in which this object resides has determined that this object is not garbage.

(2) Each object has an attribute of RECEIVEDMARK. This attribute is TRUE if a processor other than the one in which the object resides has determined that the object is not garbage and has sent a message to this object's processor indicating this condition.

(3) Each reference variable must have an attribute of MARKED. If a reference variable is copied or sent to another object in a message, this attribute is preserved in the new copy or in the message. If this attribute is TRUE, then the object that it refers to may be considered MARKED. If an object is MARKED, any or all of the references to the object may have a NOT MARKED attribute.

(4) The communication facilities must not allow messages to be hidden from all processors at any time. *If messages in transit are inaccessible to processors*, then a copy of the message must be kept by the sender until the message is known to have arrived at its destination. It is required that every message in the system be accessible to at least one processor at all times.

Qualitatively, the algorithm operates in the following manner. All of the processors are told to clear all the MARKED attributes of their objects and reference variables. When this operation is completed, all the processors are told to begin marking non-garbage objects. At first, this operation consists of scanning all the objects in each processor and marking the ones that are executable and recursively marking all the objects and reference variables that can be reached by following pointers from the executable objects. If the processor determines that an object, which resides in another processor, is to be marked a message is sent to that object, wherever it does reside, to cause it to be marked by its processor.

As long as a processor is in the mark phase, it must process incoming messages in a different manner than usual. It must mark the recipient of the message (if it has not already been marked) and it must mark any objects referred to by reference variables in the message where the reference variable is not marked. Thus, as the processors enter the mark phase, waves of set mark attributes emanate from executable objects and from objects that are in involved in communication with each other. It is assumed that the participants in an exchange of messages, and objects referred to in messages, are not garbage since they are obviously in use. This use of the normal communication between objects as part of the marking process speeds up the rate at which garbage can be collected but does not add to the message traffic. In effect, the object communication performs a double duty during the mark phase. It accomplishes the function programmed in the objects as well as identifies the objects involved as non-garbage.

When the marking of objects has finished, the remaining unmarked objects are collected as garbage. The resources belonging to these objects, their name and disk space, are released for use by new objects. The cycle is then repeated by again clearing all the mark attributes.

The processor controlling the phases of garbage collection executes the following task. The processor executing this task may be multiplexed among other tasks as well, or it may be a dedicated processor. The algorithm is described in a Simula-like syntax. Procedures such as "SendMessage" and "SendMessageToAllProcessors" are not shown in detail since they depend on the particular hardware and software communication facilities available. It is hoped that the function of the undefined procedures is self-evident.

```
WHILE TRUE DO BEGIN

    PROCEDURE WaitForAllDone;
    BEGIN
      BOOLEAN Done;
      Done := FALSE;
      WHILE NOT Done DO
      BEGIN
          SendMessageToAllProcessors("StartInterval");
          WaitUntilAllAcknowledge;
          SendMessageToAllProcessors("EndInterval");
          WaitUntilAllAcknowledge;
          Done := ANDofAllDoneFlags;
      END of While;
    END of PROCEDURE WaitForAllDone;

    SendMessageToAllProcessors("ClearAllMarks");
    WaitForAllDone;
    SendMessageToAllProcessors("EndClearAllMarks");
    WaitForAllDone;
    IF ThereIsARootObject THEN BEGIN
      REF(Processor)Root;
      Root :- ProcessorWithRootObject;
      Root.SendMessage("MarkRootObject");
    END of IF;
    SendMessageToAllProcessors("MarkExecutableObjects");
    WaitForAllDone;
    SendMessageToAllProcessors("CollectUnmarkedObjects");
    WaitForAllDone;
    SendMessageToAllProcessors("EndCollectUnmarkedObjects");
    SendMessageToAllProcessors("EndMarkingExecutableObjects");
    WaitForAllDone;
END of Garbage Collector Control Loop;
```

The loop above contains no "critical regions" and none of its operations must be "indivisible". If this task shares a processor with other tasks, the processor may be removed from this task at any point in the loop. The only effect such multiplexing may have is to reduce the rate at which garbage collection proceeds by a very small amount, providing this task receives even minimal service from the processor. The exclusive access given a processor to the objects contained in its memory simplifies the interactions between processors. The synchronization, mutual exclusion and other conditions that must be met are embedded in the sequence of message passing. The indivisible operations that must exist in such a system are those of message transmission and reception.

The messages sent to all the processors could be broadcast, if the connection medium permits it. The "WaitUntilAllAcknowledge" procedure must hold further execution until it is known that every processor has received the previous message. This operation is the primary means by which the processors and the controlling garbage collector task are synchronized. This synchronization is of a very weak nature. The acknowledgement of the processors could be detected by waiting until all the processors pulling down an open-collector TTL signal have released it, or it could be detected by the receipt of an

acknowledging message from each processor, depending on the communication facilities present. Some of the message sequences in the above loop could be concatenated into single messages but have been separated for clarity.

The "ANDofAllDoneFlags" is a hypothetical procedure which returns TRUE if the "DoneFlag" (described below) of every processor is TRUE. This function could be performed by querying each processor with an exchange of messages or with hardware, such as an open-collector signal wired to all the processors. The "DoneFlag" is defined to be valid at the time a processor does the "AcknowledgeMessage" operation and until it receives its next "StartInterval" message.

The determination of when all the marks in the system are clear or when all the garbage objects have been collected or, most importantly, when all the non-garbage objects have been marked is the mechanism that permits this algorithm to work. The most difficult question is how to determine when the marking of non-garbage objects is complete and to be assured that no more objects can be or will be marked. The collection phase cannot be initiated until the marking is finished.

The "StartInterval" and "EndInterval" messages from the controlling task delimit a span of time in each individual processor. The sequencing of the controlling task insures, despite any skew in the arrival of the messages at the processors, that a sub-interval of all the spans of time is common to all the processors in the system.

It can be said that if, during some interval of time, not one of the processors in the system marked any objects nor had any objects that were waiting to be marked, no further marking can occur in the system. When a processor receives a "StartInterval" message during a mark phase (but not while in a collection phase) it scans all its objects for any that should be marked but are not. If any objects are marked by the processor, its "DoneFlag" will subsequently exhibit FALSE. During the interval the processor may mark objects and will again record the fact if any are marked. When a "EndInterval" message is received and no objects have been marked since the interval began, the processor will again scan its objects and record any that are marked. At the end of the interval the "DoneFlag" is displayed indicating, if TRUE, that no marking was done or could have been done during the interval in that processor.

If all the processors display a TRUE "DoneFlag" at the end of an interval, then there were no objects marked in the entire system during that portion of the interval shared by all the processors. It follows that if, over the entire system, no objects were marked and no objects were waiting to be marked, then the mark phase has finished. An object must be marked to cause other objects to be marked. Therefore, if there are none to be marked and no marking has been done, there can be no further marking.

Several aspects of the sequence of messages initiated by the controlling task should be noted. The collection phase has been made a part of the marking phase. This relationship insures that any new objects created before and during the collection phase are created marked and are prevented from being collected as garbage. Otherwise, new objects could be created with a FALSE mark bit after the marking is completed but before collection, causing any such objects to be regarded as garbage. The overlapping of the mark phase with the collection phase prevents this situation.

In addition, there is what might be regarded as a spurious "WaitForAllDone" procedure inserted between the end of the clear phase and the beginning of the mark phase. This invocation serves only to insure that all of the processors have stopped clearing prior to beginning to mark. If the situation arose where some

processors were already into the next mark phase before others had recognized the end of the previous clear phase, not only would confusion result in the state of various mark bits, but neither set of processors could complete their respective phases since messages would continue to arrive with marks in an unexpected state.

A detailed description of the functions that must be performed by each processor as part of garbage collection is below. This description is shown as a message dispatch routine that intercepts and disposes of all the incoming messages of a processor. In a actual system, the mechanism associated with receiving messages from the garbage collection controller may be quite separate from the facilities used to process messages from other processors.

The procedure below would be entered when a complete message is available to the processor. Upon returning from the procedure the processor's scheduler would select other tasks for execution. In the form shown here, this procedure cannot be interrupted for the execution of objects, but may be interrupted for other tasks.

```
PROCEDURE DispatchMsg(Message); REF(Msg)Message;
BEGIN
    BOOLEAN Clearing,Marking,Collecting,DoneFlag;

    PROCEDURE MarkObject(abc); REF(Object)abc;
    IF NOT abc.Marked THEN BEGIN
       REF(ReferenceVariable)RefVar;
       abc.Marked := TRUE;
       abc.ReceivedMark := FALSE;
       DoneFlag := FALSE;
       FOR RefVar :- abc.EachRefVarInThisObject DO BEGIN
            IF NOT RefVar.Marked THEN BEGIN
               RefVar.Marked := TRUE;
               IF RefVar.Object.InThisProcessor THEN
                    MarkObject(RefVar.Object)
               ELSE RefVar.Object.SendMessage("TurnOnReceivedMark");
            END;
       END of FOR Loop;
    END of PROCEDURE MarkObject;

    PROCEDURE DoFunction;
    BEGIN
       REF(Object)Obj;
       IF Clearing THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO BEGIN
               IF NOT Obj.AllClear THEN BEGIN
                    ClearAllMarkBits(Obj);
                    DoneFlag := FALSE;
               END;
            END of FOR;
       END ELSE
       IF Marking AND NOT Collecting THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO
               IF Obj.Executable OR Obj.ReceivedMark THEN MarkObject(Obj);
       END ELSE
       IF Collecting THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO BEGIN
               IF NOT Obj.Marked THEN BEGIN
                    RecoverGarbageObject(Obj);
                    DoneFlag := FALSE;
               END;
            END of FOR;
       END of IF;
    END of PROCEDURE DoFunction;

    IF Message.Destination=GarbageCollector THEN BEGIN
       IF Message.Txt="ClearAllMarks" THEN Clearing:=TRUE
       ELSE
       IF Message.Txt="EndClearAllMarks" THEN Clearing:=FALSE
       ELSE
       IF Message.Txt="MarkExecutableObjects" THEN Marking:=TRUE
       ELSE
       IF Message.Txt="CollectUnmarkedObjects" THEN Collecting:=TRUE
       ELSE
       IF Message.Txt="EndCollectUnmarkedObjects" THEN Collecting:=FALSE
```

```
        ELSE
        IF Message.Txt="EndMarkingExecutableObjects" THEN Marking:=FALSE
        ELSE
        IF Message.Txt="MarkRootObject" THEN MarkObject(RootObject)
        ELSE
        IF Message.Txt="StartInterval" THEN BEGIN
            AcknowledgeMessage;
            DoneFlag := TRUE;
            DoFunction;
        END ELSE
        IF Message.Txt="EndInterval" THEN BEGIN
            IF DoneFlag THEN DoFunction;
            AcknowledgeMessage;
        END;
    END ELSE
    IF Message.IsObjectTransfer THEN BEGIN
        REF(Object)Obj;
        Obj :- Message.AsObject;
        IF Obj.Marked AND NOT Marking THEN BEGIN
            Obj.Marked := FALSE;
            Obj.ReceivedMark := TRUE;
        END;
        PutObjectInProcessor(Obj);
    END ELSE BEGIN
        REF(Object)Obj;
        Obj :- Message.Destination;
        IF Message.Txt="TurnOnReceivedMark" THEN BEGIN
            IF Marking THEN MarkObject(Obj)
            ELSE Obj.ReceivedMark := TRUE;
        END ELSE
        IF Marking THEN BEGIN
            REF(ReferenceVariable)RefVar;
            MarkObj(Obj);
            FOR RefVar :- Message.EachRefVar DO BEGIN
                IF NOT RefVar.Marked THEN BEGIN
                    RefVar.Marked := TRUE;
                    IF RefVar.Object.InThisProcessor THEN
                        MarkObject(RefVar.Object)
                    ELSE RefVar.Object.SendMessage("TurnOnReceivedMark");
                END;
            END of FOR Loop;
            GiveMessageToObject(Message);
        END ELSE GiveMessageToObject(Message);
    END of IF;
END of Message Dispatcher;
```

One important part of the algorithm cannot be represented as part of a message dispatch routine. This part of the algorithm must be invoked whenever a new object is to be created in a processor. It can be stated simply as follows:

```
IF Marking THEN BEGIN
    REF(Object)Obj;
    Obj :- TheNewlyCreatedObject;
    Obj.Marked := TRUE;
END of IF;
```

This provision exists to insure that all objects created in a processor while that processor is in a mark phase are created MARKED to prevent their premature collection in the next phase.

Messages representing objects that have been moved from one processor to another are accounted for in the message dispatch procedure. The only requirement placed by this algorithm on such messages is that if a marked object arrives at a processor that is not yet in the mark phase, that object becomes unmarked and acquires the attribute of RECEIVEDMARK before becoming a bona fide resident of the processor. When the receiving processor enters the mark phase, it will note the attribute of RECEIVEDMARK in the object and will mark it on the first pass. If the object were left marked upon entering a processor that was not in the mark phase, the object could acquire unmarked reference variables in the course of communicating with other objects. Since the object is already marked, these reference variables and possibly their corresponding objects may never be marked, resulting in objects being falsely collected as garbage.

In the "MarkObject" procedure, the attribute "EachRefVarInThisObject" is taken to return each reference variable associated with the object in question. Reference variables contained in unprocessed messages, or contained in an internal stack must be included as well as those that are part of the visible state of the object.

The generation of garbage by the system continues without regard for the phases of garbage collection. At any time, reference variables may be overwritten with other reference variables. When all the reference variables pointing a set of non-executable objects are destroyed, the objects become garbage. This process occurs during the mark and collection phases and at all other times as well. Objects that have been marked and subsequently become garbage will not be collected in the next collection phase. However, it is guaranteed that the next time around through the mark phase, they will not be marked and hence will be collected in the next cycle.

One refinement of the above algorithm would eliminate the clear phase. After all the unmarked objects have been collected in the collection phase, the remaining objects and their reference variables must all be marked. Thus, only the sense of the mark bits needs to be changed to consider the system cleared. A mark pass must set all the marks in the system to the same value. In the routines above the value is TRUE (presumably a one). If, instead of sending the "ClearAllMarks" message, a message indicating "InvertMarkSense" was sent, then on the next pass, a mark with a one in it would be considered unmarked rather than marked. After that pass the sense would again be inverted and so forth after each pass. This refinement has not been shown in the algorithm to preserve its readability. If this technique were adopted, a substantial fraction of the garbage collection overhead would be eliminated.

4. Simulation Results

To support the contention that the garbage collection algorithm performs as described above, a discrete simulation of its components was written and run giving every indication that it is a viable technique. The simulation was implemented in Simula using the Demos simulation package [Birtwhistle79]. A stochastic model of the executing objects was using to represent a system of running objects. The objects were given the necessary attributes and placed among a set of simulated processors each containing the garbage collection routines defined above.

The model used to represent the executing objects was a set of probabilities picked to insure that all the pathological cases of garbage collection were well exercised. In the absence of any experience or data available for concurrent programs executing on a collection of processors, the numeric values were picked to be both acceptable within the scope of experience on uni-processor systems and to be a true test of the algorithm. The resulting simulation shows that for the situations encountered using the model, the algorithm performed as expected. Some statistics were derived from the simulation but these are more a description of the simulated environment than a prediction of efficiency or performance.

The following is a detailed description of the model used for simulation.

(1) The basic time-slice interval of a processor was an average of .0167 seconds with a standard deviation of .008 seconds. The time-slices of each processor varied about the mean with a normal distribution.

(2) The probability that a given time-slice was used by a processor to service its garbage collection task was 0.50

(3) The size of an object, in terms of the number of reference variables it held was a normal distribution with a mean of 12 and a standard deviation of 10. Once created, the size of the object remains fixed.

(4) If a time-slice was used by a processor to execute objects, the number of objects "touched" in that time was a uniform distribution from 1 to 5 (inclusive).

(5) If an object was touched, the probability that it completed its execution, becoming idle to await another message, within that time-slice was 0.12

(6) The probability that it would cause a new object to be created was 0.10

(7) The probability that it would be moved to another processor if touched was 0.20

(8) The probability that it would communicate with the objects for which it had reference variables was 0.30

(9) If it communicated, the probability that any particular reference variable contained in the object or in any of its "sub" objects was transferred in a message was 0.15

Given the practical limitations of address space and processor bandwidth, no more than 36 processors could be simulated with 1800 objects between them. The machine used for the simulation was a DECsystem-2060 running Simula version 5. The simulation that produced the results below consumed approximately 10 hours of CPU time. The processors are defined to have a capacity of one

third more objects than the total number of objects divided by the number of processors, giving an average utilization of 75% among the processors. No attempt was made in the simulation to provide or maintain locality between the objects. Here again, it was thought that uniform communication between objects and hence between the processors was a more rigorous test than one with some presumed degree of locality or a presumed topology.

The simulator does not presume the existence of a "root" object. After the initial set of objects are created in the simulated system, a random set amounting to 40% of the total set of objects are set to be "executable". The simulation of the system proceeds with these objects until an equilibrium is reached with some varying percentage of 1800 objects active at any given time based solely on the simulated communication between the objects.

A check is built into the simulator to verify that the garbage collection works. Prior to each mark phase, the simulated system is stopped and a conventional garbage collection is performed to construct a list of objects known to be garbage at that instant. The objects in the list are left in place in the system and merely noted for subsequent reference. After the collection pass, the system is again stopped and the objects collected by the algorithm are compared with those noted in the list. Every object in the list must have been collected. If it was not, an error is generated since the algorithm would have failed to collect objects it should have collected. Failing to collect a garbage object would eventually cause a system to fail as the uncollected objects accumulate until they occupy all the resources of the system.

A second check makes certain that after each collection pass there are no references in the remaining objects to any of those previously collected. Again, if any object in the system is found to contain a reference to an object that the garbage collector has removed, an error is reported. Such a failure would indicate that the algorithm had falsely collected an object that was not garbage. Such an action would cause a fatal error in any real system.

At no time during the execution of the final simulator were any such errors reported. While this fact is not a rigorous proof that the algorithm is error free, it inspires a high degree of confidence that it does perform as expected. The simulated system is known to produce all the pathological pointer structures that might be expected to trouble the algorithm. Also, the built-in skew between the processors in the rate at which they poll for messages from the the controlling garbage collector task insures that synchronization problems, if any, arising from the differing states of the processors would be detected. Interprocessor interactions, such as object transfers and the transfer of reference variables in messages, are amplified in the simulator to aggravate any possible weaknesses in the algorithm. No weaknesses have been detected by simulation.

Statistical Data Taken from Simulation

Item	Observations	Average	Std.Dev.	Minimum	Maximum
Number of Objects in Existence	141754	1613.631	271.287	1031.000	1800.000
Number of Executable Objects	226594	703.869	54.122	504.000	845.000
Number of Mark Repetitions per Cycle	114	4.044	0.245	3.000	5.000
Time Required per Cycle	114	2.607	0.163	2.255	3.023
Lifetime of Objects	70877	6.816	8.088	1.205	199.451
Number of Objects Collected per Cycle	114	621.728	50.368	504.000	769.000

Histogram of Number of Objects Collected Each Cycle

	Objects	Cycles	Freq	Cum	
1	0	0	0.00	0.00	I
2	75	0	0.00	0.00	I
3	150	0	0.00	0.00	I
4	225	0	0.00	0.00	I
5	300	0	0.00	0.00	I
6	375	0	0.00	0.00	I
7	450	1	0.01	0.88	I*
8	525	41	0.36	36.84	I**********
9	600	53	0.46	83.33	I**********
10	675	17	0.15	98.25	I**********
11	>750	2	0.02	100.00	I*

Histogram of Object Lifetimes

	Age	Objects	Freq	Cum	
1	0	21028	0.30	29.67	I******************************
2	2	3967	0.06	35.27	I******
3	4	18995	0.27	62.07	I**************************
4	6	10811	0.15	77.32	I***************
5	8	5841	0.08	85.56	I********
6	10	2392	0.03	88.93	I***
7	12	1251	0.02	90.70	I**
8	14	1717	0.02	93.12	I**
9	16	1164	0.02	94.76	I**
10	18	781	0.01	95.87	I*
11	20	345	0.00	96.35	I.
12	22	327	0.00	96.81	I.
13	24	346	0.00	97.30	I.
14	26	293	0.00	97.72	I.
15	28	210	0.00	98.01	I.
16	30	126	0.00	98.19	I.
17	32	154	0.00	98.41	I.
18	34	184	0.00	98.67	I.
19	36	119	0.00	98.83	I.
20	38	108	0.00	98.99	I.
21	40	61	0.00	99.07	I.
22	42	85	0.00	99.19	I.
23	44	61	0.00	99.28	I.
24	46	70	0.00	99.38	I.
25	>48	441	0.01	100.00	I*

The data tabulated above shows the characteristics of the running system derived from the model chosen for simulation. It also shows how the garbage collection algorithm performs in this system. Of note is the maximum of 5 iterations required to mark all non-garbage objects. It is also worth noting, that in this system, approximately half of the 1600 objects in existence are executable at any given time. The "notch" in the lifetime histogram at 2 seconds is due to the cycle time of the collection process. With a cycle time of 2.6 seconds, any objects created during the mark phase of a cycle must wait until the next full cycle to be collected if they are made garbage. This effect skews the graph of what would otherwise be a Poisson distribution. Other interesting aspects are: An object has a 90% probability of being collected as garbage within 4 cycles of the garbage collector. Typically, one third to one half the existing objects are collected each cycle. These figures indicate a rapid turnover in the objects and thus a rapidly changing data structure.

5. Performance Analysis

The simulation, despite the care with which the model parameters were chosen, cannot give more than clues to how the algorithm might perform in a

real system. The characteristics of an actual set of objects in a real system, communicating with each other in some topology, executing and manipulating pointers in some manner and migrating between processors, is unknowable *a priori* and depends as much on the application of the system as on the system itself. However, this garbage collection algorithm, while it is intended to operate in such an environment, can be compared with conventional garbage collectors on uni-processor systems. In addition, it is important that the algorithm's performance scale well as the number of processors in the system increases.

The conventional garbage collection program running on a single processor system must make at least two passes across the data structure it manipulates. It must first follow every path from known non-garbage objects to mark every object that can be reached. It must then make a pass sequentially through the objects linking the garbage objects together or compacting the non-garbage objects into a contiguous area of memory. This second pass has a complexity of O(N) where N is either the number of garbage objects or the number of non-garbage objects depending on whether the linking or compaction is to be done. If N is defined to be the total number of objects, then O(N) is an upper bound for the complexity of the second pass. In the first pass, every pointer must be followed to the object it points to, and if the object has not already been visited, then it is marked and all of that object's pointers must be visited. Thus the complexity of this pass is $O(N_{ob}+N_{pt})$ where N_{ob} is the number of non-garbage objects and N_{pt} is the number of pointers contained in those objects. If the worst case is assumed where none of the objects are garbage, the complexity for both passes together is $O(2N+N_{pt})$. Further, if the average number of pointers contained in an object is M then we have O(N(2+M)). It should be noted that both N and M are bounded by the address space of the machine and M>1.

For an individual processor in a multi-processor machine using the algorithm presented here, the same observations can be made concerning each pass of the combined mark and collect phases. If the data structure being collected were contained wholely within that processor the complexity of one pass would be O(N(2+M)) just as for the conventional garbage collector. However, the messages that result in the marking of objects in other processors and the migration of objects between processors cloud the issue. Owing to the method by which the completion of a phase is determined, both the mark pass and the collection pass are run a minimum of twice over the objects in the processor. It should be evident that the second and any subsequent scans by these routines will require less computing since most, if not all, the objects will have been touched on the first scan. If we assume the worst, this doubling of the scans increases the complexity to O(2N(2+M)).

In general, the passing of pointers and objects between processors and the tracing of pointer paths between processors will increase the number of scans (the number of times the loop in "WaitForAllDone" is executed) beyond the minimum of two. The volume, pointer content, locality and speed of the message traffic in the system all affect this number. Both the applications being run on the system and the hardware communication facilities will determine how long unmarked or uncollected objects can exist in the system once the mark phase has begun and thus how many scans (of "DoFunction") will be required to catch all the moving objects and reference variables. Simulation of the system where communication is very fast, locality is non-existent and pointer content of messages is high shows that no more than 5 scans are needed and typically 4 are sufficient in a system of 36 processors. For now, a number can be defined which is the average number of such scans required for each complete cycle of the garbage collection task. If this number is represented by X then the complexity of one cycle in one processor becomes O(XN(2+M)), where N is the number of

objects held by the processor and M is the average number of reference variables in an object.

We can now compare the complexity of collecting garbage in a system with one processor versus a system with many where this algorithm is used and where the total number of objects is the same. The complexity of one garbage collection pass in a single processor system remains $O(N(2+M))$. However, the same number of objects N distributed over P processors gives a complexity of $O(XN(2+M)/P)$. Therefore, whenever the ratio $X/P < 1$ or when $P > X$ the multiprocessor system exhibits less overhead in garbage collection than its uniprocessor counterpart, all other factors being equal. If 4 is used for X we find that 4 or more processors running the garbage collection algorithm presented here perform better than a single processor. Real values for X will have to await the construction of such machines and the accumulation of experience with their use.

Aside from the overhead incurred by garbage collection, there is a second important measure of the performance of an "on-the-fly" collection algorithm. In conventional, dynamic programming environment with a single processor, the sequential garbage collector is invoked when available memory for new objects gets low or becomes exhausted. In this type of system, the rate at which garbage is collected is made equal to the rate at which it is generated on a short term basis because the garbage collection occurs on demand. The "on-the-fly" algorithm presented here cannot be invoked on demand, but instead proceeds to completion at its own rate and then starts over. On a long term basis, it must also collect garbage at the same rate it is generated. In the short term, if system resources get low, the creation of new objects will have to wait for the completion of the current collection cycle when the resources held by garbage objects are made available.

The number of cycles the collection algorithm can perform per unit time will determine the performance of systems that run near the limit of their resources. When the resources get very low, objects which try to create new objects will be held from executing and the processors may spend inordinate effort attempting to move its objects to other processors. Of course, as more and more objects are suspended, additional processor bandwidth is available to perform garbage collection functions causing the current cycle to complete sooner. Compared to the conventional garbage collector, the main factor in determining the speed with which cycles are completed is the factor X defined above. If X is large enough to permit short term depletion of resources in the system, then time devoted to object execution will decrease and that devoted to garbage collection tasks will increase in inverse proportion to the available resources. Providing a thrashing condition can be avoided with all the processors attempting to foist excess objects off on each other at the same time, this effect provides negative feedback to balance the effort used to garbage collect versus that used to execute objects.

6. Implementation Considerations

At a minimum, the integration of this garbage collection algorithm on a set of processors does not require any additional hardware beyond that which exists for the normal communication between objects. However, in some situations, using the existing facilities may not be desirable. Two capabilities make the communication between the controlling task and all the processors much more efficient and convenient. These are a broadcast capability and a wired-AND

capability.

As can be seen from the description of the algorithm, the "SendMessageToAllProcessors" operation is an important one to controlling the phases of the collection process. It would be undesirable if this operation must be implemented as the sending of an individual message to each processor in turn. If the number of processors is large, the communication bandwidth used in doing so may be unacceptable. Also, the skew introduced by the widely varied times at which the processors receive the messages will increase the overhead (i.e. the factor X will increase) and slow the rate of garbage collection. Clearly, for all but small systems, those with fewer than 20 processors, a broadcast capability is necessary.

The wired-AND capability is helpful in performing the "ANDofAllDoneFlags" function in the controlling task. Without the hardware to assist in this function, each processor will be required to respond with a message containing the state of its "DoneFlag". Again, the time and bandwidth used in sending such messages will have a detrimental effect on the performance of the system. The broadcast capability, if it exists, would be of no help since each processor has an individual "DoneFlag". However, since the controlling task is only interested in whether or not all of the "DoneFlags" are TRUE, a single wire using either an open-collector or open-emitter technology could be used to perform the logical AND on the wire. The "WaitUntilAllAcknowledge" function could be implemented in the same manner. A set of such signals, connected to every processor, would also make the acknowledgement of garbage collector messages by the processors much more convenient and efficient.

If a sufficient set of signals is connected to all the processors and to the controlling task, the controlling task can be reduced to a small finite state machine. A simple analysis shows that the necessary functions could be provided with less that 8 separate signals and possibly with as few as 4.

7. Scaling of the Algorithm

Of key importance, is the ability of the algorithm to scale as the number of processors and objects increase. Specifically, is the increased computing bandwidth obtained by adding processors to the system still available as the garbage collection operates over a larger number of processors and objects? It is evident from the description of the algorithm that global communication is required between the controlling task and every other processor. In many systems, such global communication prevents the systems from growing effectively beyond some limit imposed by the cost or the delay introduced by such communication facilities.

In the case where no broadcast facilities are present or where there is no hardware support for the "ANDofAllDoneFlags" function, there is clearly a delay in communication that grows linearly with the number of processors in the system. For small systems this delay may be tolerable but in order to build large systems consisting of thousands of processors, the broadcast and wired function facilities must be presumed.

If, as proposed above, a small number of signals are to connect every processor in parallel to the controlling task (or finite state machine), then all of the processors of the system are to be connected in a linear array. There must exist some other facility by which the objects communicate between processors. the simplest such connection is also a linear array or line such as an Ethernet

[Metcalfe76]. Any other connection must be topologically more complex. Thus the connection of all the processor to a multi-conductor cable is on the same or on a simpler order than the network that must exist to connect the processors for normal communication.

The next difficulty with the global communication is that of fan-out and delay. Clearly, no logic technology provides the means to directly connect thousands of loads or sources to a single conductor as might be desired. However, due to the simple manner in which the processors and the controlling task communicate, a hierarchy of buffers can be constructed in a tree structure to limit the fan-out and fan-in of the components. If parts are used permitting a fan-out and fan-in of 16, then a system of 64K processors would have only four levels of buffers. If the buffers introduce a delay of 25 nanoseconds each and the propagation delay of the transmission lines between them is 900 nanoseconds (the delay seen across 600 feet of wire) then a very conservative estimate of the maximum delay between the controlling task and the processors is one microsecond. For the purposes of the garbage collection, this figure is so small it can be disregarded. The delay is proportional to the logarithm of the number of processors and will thus remain very low for even greater numbers of processors beyond the capability of current technology to package and power such systems.

The number of iterations of marking that must be executed by the processors before completion of the mark phase (denoted by the factor X above) must be investigated for large numbers of processors. To be a viable collection algorithm, the number of iterations must remain small as the number of processors is increased.

A number of processor configurations were simulated to test the effect of increasing the number of processors on the average time required to complete a garbage collection cycle. In each case the average number of objects per processor was maintained at approximately 25. All other factors in the model were held at the values described above.

Simulation Results (with Message Polling)

Number of Processors	Average Cycle Time	Average Number of Repetitions per Cycle
1	0.52	2.0
2	0.78	2.4
4	1.19	3.1
8	1.62	3.3
16	1.91	3.2
32	2.33	3.3
64	2.72	3.5

Simulation Results (with Message Interrupts)

Number of Processors	Average Cycle Time	Average Number of Repetitions per Cycle
1	.260	2.00
2	.290	2.85
4	.303	3.07
8	.309	3.27
16	.316	3.47
32	.333	3.96
64	.337	4.11

The average cycle time figures are computed in seconds. Because the timing in the simulator has no basis in the hardware of a real system, the absolute value of the numbers cannot hold much meaning. However, the relationship between these numbers is indicative of how the algorithm scales as the number of processors in increased.

The simulator maintains no policy of locality to control the placement of objects among processors. Thus, in cases where a non-executing path of pointers and objects is strung across many processors and an object in the path is referenced by a non-garbage object, it may take several repetitions for the marking routines to follow the path through all the processors.

The average time required to complete a garbage collection cycle is observed to rise at a linear rate with the logarithm of the number of processors. The time to complete the cycle is a function of both the number of repetitions required in the cycle and the time required for all the processors to acknowledge messages, where polling is simulated. If processors are interrupted upon the receipt of a message from the controlling task, then there is essentially no skew between the processors and minimum delay in the acknowledgment of the message. The data tabulated for the simulation with interrupts is plotted in figures 1 and 2. The short vertical bars represent one standard deviation of variance about the average shown in the table.

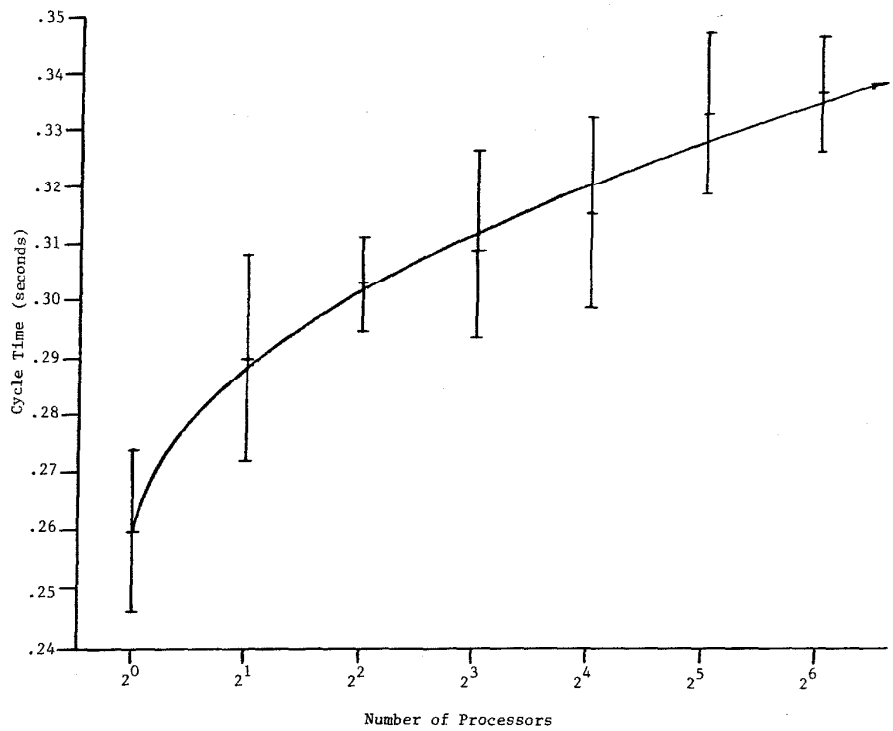

Figure 1

Cycle Time vs. LOG(Number of Processors)

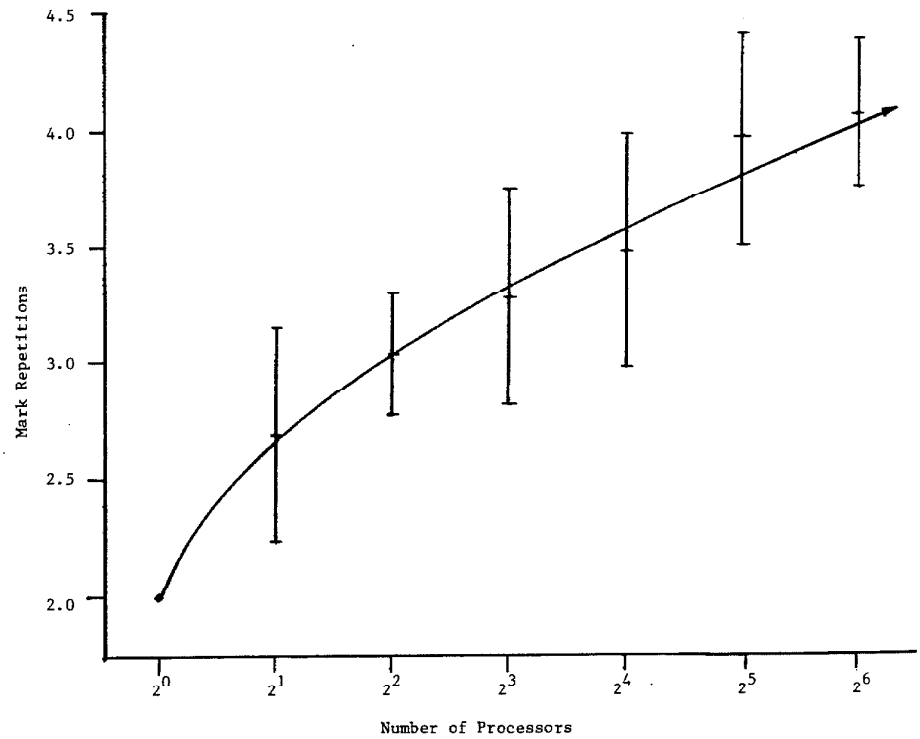

Figure 2

Mark Repetitions vs. LOG(Number of Processors)

The figures above show that in the worst case, the number of garbage collection cycles completed per unit time decreases linearly with the logarithm of the number of processors. The average number of repetitions required per cycle may grow linearly with the logarithm of the number of processors as well but there is some evidence that it may roll off to a value less than 5. In the best

case, the average number of cycles per unit time also becomes a constant. Where the performance of a particular system falls between these two cases will be determined by the communication structure of the machine and the degree of locality present in the objects and processors. However, the worst case is seen to result in a performance of the garbage collector proportional to the inverse of the logarithm of the number of processors. If one merely extends the simulation figures to 64K processors, we see that the cycle time of the algorithm would be about 6.6, only 60% slower than 64 processors. The average number of objects per processor is presumed to remain constant, meaning that the 64K processor system contains 1,000 times the objects contained in the 64 processor system. If the relationship is as suggested by these numbers, the algorithm can be said to scale very well as the size of the system is increased.

8. Summary

A garbage collection algorithm has been presented which satisfies the needs of systems consisting of large numbers of processors. The correct operation of the algorithm has been demonstrated by simulating its operation. The unit of collection, the object, while a construct of programming languages, can be applied broadly to a wide range of systems, including conventional file systems and database systems.

The collection algorithm benefits from, but does not require, hardware communication facilities dedicated to the task of garbage collection. If these facilities are present, the speed of the garbage collection decreases in proportion to the logarithm of the number of processors, in the worst case. The introduction of techniques to improve the locality of reference among the objects in the processors will improve upon the already acceptable scaling characteristics of the algorithm.

The ability to collect garbage from data structures distributed among many processors in an efficient manner is a necessary ingredient to the use of very large distributed machines for general applications. Systems which provide for concurrency by connecting multiple processors to a single memory are necessarily limited in both size and performance. In the environment considered here, processors are the sole masters of objects resident in their private memories and communicate with other processors by passing messages through a communication network. The algorithm presented here will support the distribution of data and computation across a very large number of such processors without introducing more overhead computation than conventional collection algorithms require on existing systems.

Bibliography

[Almes80] Guy T. Almes, "Garbage Collection in an Object-Oriented System", Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, June 1980.

[Arnborg72] Stefan Arnborg, "Storage Administration in a Virtual Memory Simula System", *BIT: Nordisk Tidskrift for Informationsbehandling*, Volume 12, pp.125-141, 1972.

[Backus78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM*, Volume 21, Number 8, pp. 613-641, August 1978.

[Birtwhistle73] G. M. Birtwhistle, O-J Dahl, B. Myrhaug, K. Nygaard, *Simula Begin*, Petrocelli, New York, 1973.

[Birtwhistle79] G. M. Birtwhistle, *Discrete Event Modelling on Simula*, MacMillan, June 1979.

[Collins60] George Collins, "A Method for Overlapping and Erasure of Lists", *Communciations of the ACM*, Volume 3, pp.655-657, December 1960.

[Deutsch76] L. Peter Deutsch and Daniel G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector", *Communications of the ACM*, Volume 19, pp.522-526, September 1976.

[Dijkstra78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation", *Communications of the ACM*, Volume 21, pp.966-975, November 1978.

[Gries77] David Gries, "An Exercise in Proving Programs Correct", *Communications of the ACM*, Volume 20, pp.921-930, December 1977.

[Ingalls78] Dan Ingalls, "The Smalltalk 76 Programming System: Design and Implementation", *Proceedings of the Fifth ACM Conference on Principles of Programming Systems*, pp. 9-16, January 1978.

[Jones73] Anita K. Jones, "Protection in Programmed Systems", Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, 1973.

[McCarthy60] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine", *Communications of the ACM*, Volume 3, pp.184-195, April 1960.

[Metcalfe76] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Volume 19, pp.395-403, July 1976.

[Ritchie74] D. M. Ritchie and K. L. Thompson, "The UNIX Timesharing System", *Communications of the ACM*, July 1974.

[Steele75] G. L. Steele, "Multiprocessing Compactifying Garbage Collection", *Communications of the ACM*, Volume 18, pp.495-508, September 1975.

[Thornton70] J. G. Thornton, *Design of a Computer: The Control Data 6600*, Scott Foresman, 1970.

[Van Wijngaarden69] A. Van Wijngaarden, ed. B. J. Maillous, J. Peck and C. Koster, *Report of the Algorithmic Language ALGOL 68*, Springer-Verlag, 1969.

[Wadler76] Philip L. Wadler, "Analysis of an Algorithm for Real Time Garbage Collection", *Communications of the ACM*, Volume 19, pp.491-500, September 1976.

[Wulf72] William A. Wulf and C. G. Bell, "C.mmp- A Multi-Mini-Processor", *Proceedings of the 1972 Fall Joint Computer Conference*, pp.765-777, American

Federation of Information Processing Societies, 1972.

[Wulf80] William A. Wulf, Samuel Harbison and Roy Levin, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1980.

[Yu81] Kwang-I Yu, "Communicative Databases", Ph.D Thesis, Computer Science Department,California Institute of Technology, 1981.