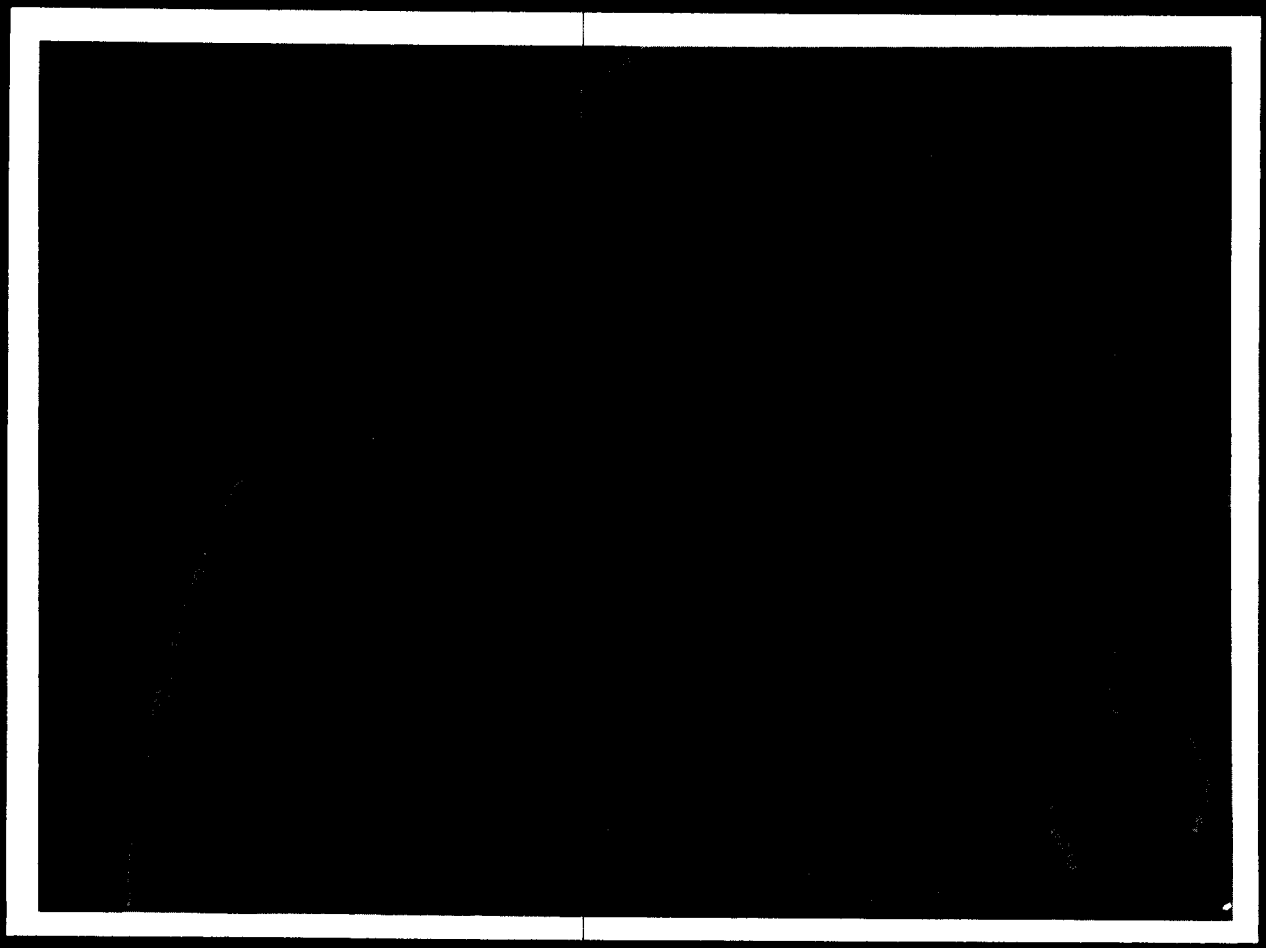
ROBOTICS and EXPERT SYSTEMS - 1985

Proceedings of ROBEXS '85

The First Annual Workshop on Robotics and Expert Systems
NASA/Johnson Space Center
June 27-28, 1985



SPONSOR:

Robotics & Expert Systems Division

CO-SPONSORS:

Clear Lake - Galveston Section and District 7
of the
Instrument Society of America

0-87664-872-3/85/081-087/$0 + .50PP

EFFICIENT REAL-TIME GARBAGE COLLECTION FOR LISP

Robert L. Shuler, Jr.
Electronics Engr.
Spacecraft Software Division
NASA Johnson Space Center
Houston, TX

ABSTRACT

A method is presented for identifying garbage in a LISP system and adding it back to the free list at the time it is created, rather than at some later time. It is shown that this technique, a modification of reference counter garbage collection, (1) successfully corrects deficiencies in earlier similar systems, (2) can be efficiently implemented in hardware, (3) places essentially insignificant constraints on the LISP system, and (4) provides the performance characteristics needed to build effective real-time systems, virtual memory systems, and systems with very large main memory.

A LISP interpreter using this system is analyzed and compared to a conventional interpreter. The possibilities of re-programming existing LISP machines are explored, as well as considerations for designing a new machine optimized for this type of garbage collection. The possible use of this technique in systems limited to 32 bit word sizes (while allowing 32 bit data items) is discussed.

INTRODUCTION

List processing finds applicatons in operating system, database, artificial intelligence, and other software where information with dynamically varying structure must be represented. This paper is principally concerned with the list processing language LISP and similar systems for manipulating linked lists with fixed cell sizes, and with artificial intelligence applications requiring conversational or real-time responsiveness, although generalization to other systems should be possible. For a summary of various applications and languages in this area consult Rich (1).

List processing is the manipulation of linked lists consisting of cells, each of which contains two pointers. The first pointer, called the CAR, points to either a sub-list or a fundamental data item (atom). The second pointer, called the CDR, points to a continuation of the list or to NIL. Atoms, once introduced, are permanent. They may be bound to a value, which may be another atom (symbol or number) or a list. New lists are constructed by allocating vacant cells from a free list, and placing in them pointers to existing lists, parts of lists, or atoms. Topology of existing lists is not normally modified, and thus several lists or atoms may reliably refer to the same underlying list fragment as part of their value without having to make their own copy as would be done in conventional languages. All accessible cells may be reached by tracing down either a list bound to an atom, a list bound to a stack entry, or the free list. As the values of atoms are changed, some cells become inaccessible. Identifying these cells and adding them back to the free list is called "garbage collection." Winston and Horn (2) provide a good introductory description of both LISP and garbage collection.

Mark and Sweep Strategies

In a survey by Cohen (3), garbage collection strategies are classified as two main types. "Mark and sweep" strategies trace down lists from the base atoms to mark all accessible cells, then sweep through memory to find all unmarked (and therefore inaccessible) cells and link them back onto the free list. This is most simply done while list processing is halted, but more complex multi-pass "on-the-fly" variations have been devised that can operate in parallel with list processing. Nevertheless, if the list processing operations demand free cells faster than the collector locates them, processing will halt while cells are located. Such behavior may be frequent and/or of long duration, and is unacceptable for real-time or conversational systems. Adding more memory paradoxically makes the problem worse by increasing the space in which the collector must search.

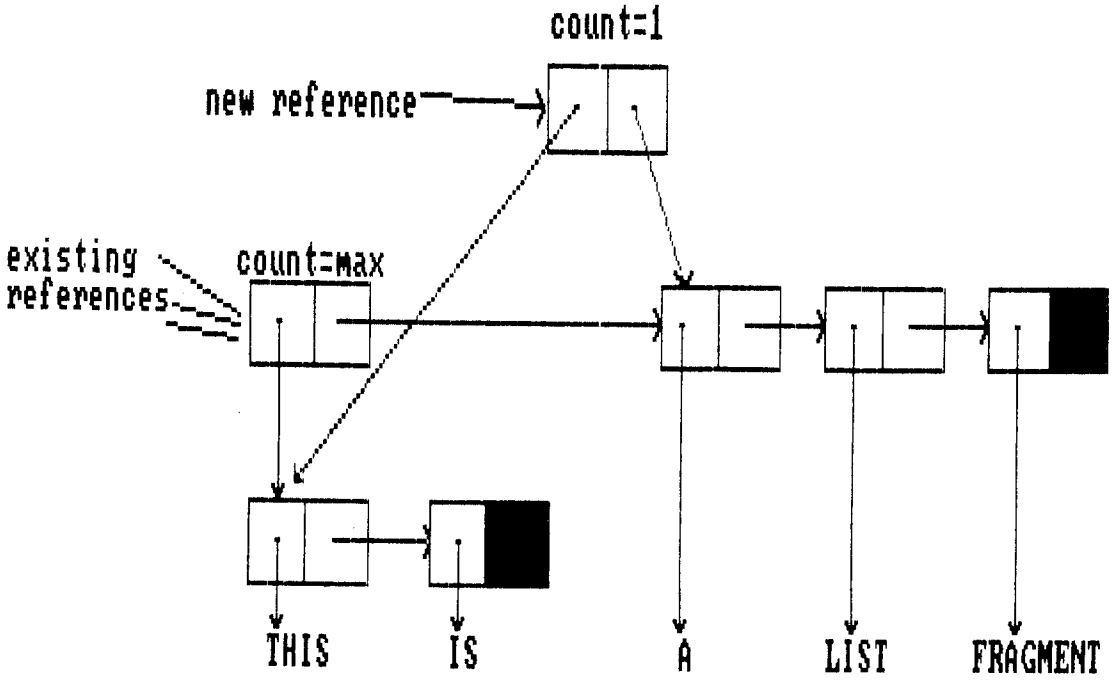
Reference Count Strategies

Reference count strategies keep a counter with each cell which is incremented each time a new reference (pointer) is created to that cell, and decremented each time an old reference is deleted. When the counter becomes zero the cell can be immediately added back to the free list. Reference counters have been popular for use with operating system data structures where their overhead is a small percentage of processing; but, in a LISP type environment the counter would have to be as large as a pointer (increasing storage requirements by 50%), and significant extra memory cycles would be required to update them since references are created and destroyed continually. Another problem with reference counters is that they are not effective in reclaiming "cyclic" list structures. The observation that reference counters usually remain small has led to hybrid schemes which use small counters, with mark and sweep collection only occasionally invoked to collect inaccessible cells whose reference counters have overflowed (and thus could not be decremented). This makes the delays less frequent, but does not eliminate them.

It is the purpose of this paper to describe two variations of reference counter garbage collection which solve the problems mentioned above and provide a good alternative to mark and sweep collection for real-time and conversational applications. The modifications are so effective that they may be preferable to mark and sweep even in non-critical applications.

COPY-MODIFIED REFERENCE COUNT STRATEGY

The copy-modified reference count strategy is based on the observation that for normal list processing operations the properties of a list depend only upon its topology, and not upon the actual cells used to represent that topology. When a reference is created to a cell whose reference counter is already at the maximum value, then the cell is copied and the reference is made instead to the new cell, which is given an initial reference count of one. Figure 1 illustrates this situation. Of course, copying the cell creates a new reference to the cells indicated in the CAR and CDR of the copied cell. If the maximum allowed reference count were one, or if all the cells in a data structure were already at the maximum reference count, then adding a new reference to the structure would result in copying the entire structure much as a more conventional language would do.



value of NEW & EXISTING lists: ((THIS IS) A LIST FRAGMENT)

Figure 1. COPY-MODIFIED LIST STRUCTURE AFTER OVERFLOW

Advantages of Copy-Modified Strategy

This strategy does not change in any way the appearance of lists, so that the bulk of logic in a list processor is unaffected. Other than the action taken on overflow, it is identical to a standard reference counter strategy. Reference counters can, however, be made arbitrarily small without risk of overflow. Small counters not only take up less space in memory, but lend themselves to various schemes to reduce the overhead of updating the counters.

Limitations of Copy-Modified Strategy

Certain operations in LISP, which may be characterized as "list splicing" operations, modify directly the values of the CAR and CDR in a cell. The EQ operation tests directly for equality of absolute addresses of two cells. Neither of these give predictable results when the underlying structure "may or may not" contain copied cells. Winston and Horn (2) do not consider use of these operations to be desirable programming practice. The author's own experience is that use of such operations, even in small programs by experienced persons, often results in obscure errors; thus, use of a garbage collection strategy which makes their use not possible should not be considered too much of a drawback.

As with any reference count strategy, cyclic structures cannot be reclaimed. This is because a cyclic structure "refers to itself" at some point, and thus will have a count of at least one even though not bound to an atom or stack entry. Cyclic structures do not naturally occur in LISP, and may be created only through list splicing. The application of many LISP operations to a cyclic structure will result in apparently infinite operations or other obscure errors. Problems whose logic demands something of a cyclic nature are normally set up with one or more atoms at appropriate points within the "cycle," which avoids all the above mentioned difficulties. Thus, this limitation should not be considered too much of a drawback either.

Implementation and Verification

An interpreter (written in PASCAL) for the MACLISP dialect described by Winston and Horn (2) was available to serve as a testbed for this collection strategy. Modification began by adding a reference counter to each cell, and by defining the following recursive procedures:

ADD_REF (PTR)

DEL_REF (PTR)

ADD_REF performs the operation of incrementing the counter for the cell referenced by its argument PTR. If the counter cannot be incremented, it performs the allocation of a new cell, copies the old cell contents to the new cell, updates PTR to point to the new cell, and finally recursively invokes itself to ADD_REF to the CAR and CDR of the new cell.

DEL_REF decrements the counter of the cell, and if zero it adds the cell back to the free list and recursively does a DEL_REF on the CAR and CDR of the CELL. Of course, if the thing being pointed at is an atom, not a cell, both routines are "no-operations." In this particular implementation, numeric values, although conceptually atoms, are dynamically allocated from the same memory space as cells, and are treated like pointer cells by ADD_REF and DEL_REF, except that no attempt is made to operate on their CAR and CDR's.

The next step was to identify within the interpreter each point at which a reference was being created or destroyed, and invoke the appropriate ADD_REF or DEL_REF function. This proved to be about as difficult as the original operation of making sure all intermediate results were "known" to the mark and sweep collector, except that errors were much easier to detect with the reference counter system, which immediately reallocates a cell that is erroneously released, rather than reallocating it at a much later time. During this process it became convenient to define two non-primitive routines to perform internal CAR and CDR operations in such a way as to delete the reference to their argument while adding a reference to the result.

After modification the interpreter was operated with a small-to-medium sized expert system which uses most of the LISP operations. This verified correctness of the concept and implementation, and provided some performance information which is presented in a following section.

COUNT-EXPANSION STRATEGY

The count-expansion strategy uses small reference counters as does the copy-modified strategy, but when more references are desired than the small counter can represent, the counter is expanded. This is accomplished by obtaining a new cell from the free list, and storing a link to the new cell in the old cell. There is space for two pointers in each cell, one of which in the first cell is used to store the link. Two of the remaining pointers are used to contain the CAR and CDR values of the original cell, and the third is used to contain an expanded reference counter. These may be allocated in any way desired, but placing the CAR and CDR in the new cell

and retaining the reference count in the first cell is most compatible with alternate cell formats, such as using a cell to store numeric atoms instead of pointers. This scheme is diagrammed in Figure 2.

Advantages of Count-Expansion

The original cell remains the principal point of reference, retaining the same absolute address for all references derived from the particular piece of list topology in question. Thus list splicing operations may be used (by the brave or the foolish). It is possible a large number of cells might have to be expanded, but the expansion is limited to one additional cell. In a copy-modified strategy a single cell might be copied many times.

Drawbacks of Count-Expansion

All the list manipulation logic in the list processor must now deal with two significantly different cell formats, and the overhead of the logic to interrogate and compensate for this fact. All list operations involving the cell will require extra memory cycles to access the remotely located second part.

Because of the additional complexities, count-expansion has not been implemented by this author. The performance statistics reported in a following section for the copy-modified strategy should apply directly to the count-expansion strategy also.

CONSIDERATIONS FOR HARDWARE IMPLEMENTATION

Greenblatt, et. al. (4), describe a hardware architecture for a LISP machine based on a general purpose 32 bit micro-programmable processor. Either copy-modified or count-expansion garbage collection appears compatible with most of the basic design of this machine.

In a pointer word, bits are allocated as follows:

24 bit address
5 bit data type
2 bit CDR coding
1 bit not mentioned (probably garbage collection flag)

A two valued reference counter can be obtained by merely re-using the no-longer needed garbage collection bit. Larger reference counters would require sacrificing bits from one of the other fields, possibly data type, or the storing of the counter in some other place, such as a special array of counters. Data types would refer to the item referenced in the address field, but the reference counter would relate to the specific cell in which it was contained. Every pointer to an entity would thus contain information about the type of that entity, but the reference count would be attached to the entity

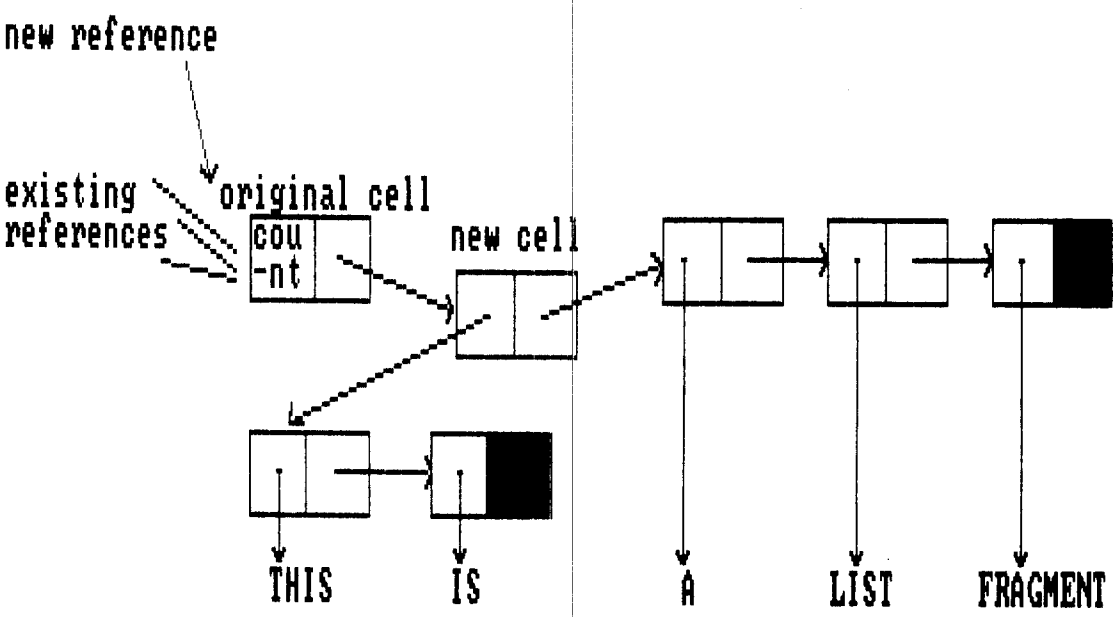


Figure 2. COUNT-EXPANDED LIST STRUCTURE AFTER OVERFLOW

itself. Primarily list cells and other complex data items would have reference counters. Atoms, of course, are permanent and do not need counters. Numeric constants could be treated as atoms, but it is usually desirable to treat numbers resulting from computation as dynamically allocated entities, like lists. If such a number requires a 32 bit format, it can be assumed that its reference count is always one, thus it will always have to be copied in a copy-modified strategy (which is comparable to what conventional non-LISP machines do), and when its one reference is deleted it can always be added back to the free list. In a count-expansion strategy it would always have to be expanded, which would take some extra storage.

LISP Machine Modifications

LISP Machine, Inc. (5) produces a commercial machine very similar to the one described by Greenblatt. It supports user re-definition of the microcode and it or a similar machine could be conceptually re-programmed to implement one of the garbage collection strategies described herein. Unlike various mark and sweep strategies, however, all the microcode which manipulated lists with the possibility of adding or deleting references would have to be examined and modified, just as it was in the above-mentioned interpreter. Any changes in basic data or control field formats would also require such global change. Presumably some logic devoted to making sure the garbage collector was aware of all in use cells (usually through special stack entries) could be eliminated, along with the mark and sweep collector itself.

Further steps to increase efficiency would require caches of some sort to reduce the overhead of counter increment and decrement, and could not likely be implemented via microcode changes alone. Cohen (3) points out that in LISP most cells are only temporarily needed, and quickly become garbage which could be added back to the free list. This indicates performance could be greatly enhanced by a cache with some sort of delay feature, whereby main memory is not updated on each write operation, thus allowing cells to make a complete loop from the free list, into use, and back onto the free list with only an initial and final main memory access. In fact, using a free list with a last-in-first-out nature would result in a pool of free cells being maintained in the cache and being utilized over and over with often no main memory accesses for a usage cycle. Various cache architectures would need to be studied for effectiveness. It might prove more reasonable to use a small, purely associative cache in which only least recently used cells were written to memory (to make space for new ones), rather than more traditional block-mapped architectures.

PERFORMANCE ISSUES

The most critical issues affecting the feasibility of either of the collection strategies presented here are the relative fraction of references which result in overflows (and thus the time spent in exceptional processing, as opposed to the routine of increment & decrement), and the relative fraction of memory which becomes devoted to storing copied or expanded cells. These must be evaluated as a function of counter size. Whatever optimum value is found can then be compared with the time spent performing mark and sweep collection and with the extra storage which such a strategy must provide in order to operate efficiently.

Results of Interpreter Measurements

The above mentioned interpreter was easily equipped to gather the statistics presented in Table 1. The application from which these data were gathered was the same expert system which was used to validate modifications to the interpreter, and so the data should be considered an indication needing further support, not an exhaustive sampling of many applications. It is evident from these data that a relatively small counter size of only two bits would result in less than 10% of the system resources being devoted to garbage collection. Even one bit counters would be feasible. This can be compared with results presented by Hickey and Cohen (6) which indicate that about 75% of the system processing power is needed by on-the-fly mark and sweep strategies to guarantee that no cycles will occur in which the list processor must wait on the collector. The performance of reference count strategies is significantly obtained without resorting to the complexity of parallel processors performing the collection. This advantage in simplicity could be very important when imbedding symbolic processing in small microprocessors.

Characterization of Frequent Overflows

In examining the raw data from this interpreter, it became apparent that perhaps only 1% or so of the cells were accounting for 80% to 90% of the overflows. Logic was added to print out the nature of such cells, and they turned out to be, without exception, numerical constants. The two greatest offenders were zero and one. These were part of the original program, and as the program executed through a few hundred thousand steps, they became incorporated into list structures at several hundred different places. In retrospect, this seems like reasonable behavior, probably typical of a wide

Table 1. OVERFLOW OVERHEAD STATISTICS

	# ADD_REF's	% of References which Generated Overflow			
	# CELLS	% of Cells which are Copies			
After Program Load	3308	8.6%	9.9%	-	-
	7777	5.6%	6.9%	-	-
After Program Execution	174189	3.4%	5.0%	-	-
	17953	5.7%	7.5%	8.5%	19.2%
Maximum Reference Counter Value →		15	7	3	1

variety of programs, and does not seem to be particularly inefficient.

Memory Size Issues

An important aspect of any reference counter collection scheme is that its performance depends only on the rate at which the list processor demands cells and creates garbage. Mark and sweep strategies, on the other hand, depend heavily on the size of the space being scavenged, growing less efficient as the space becomes larger. This has resulted in various complex partitioning schemes as described by Greenblatt (4), aspects of which must be taken into account by the applications designer.

A second important aspect of reference counter schemes is that by being able to identify and re-utilize garbage immediately, they can be designed to keep the memory space occupied by a program more compact. This is critical in virtual memory systems where locality of reference can make or break system performance.

SUMMARY AND CONCLUSION

Methods have been presented for solving the problems of counter size, counter overflow, and counter increment/decrement overhead for reference count garbage collection, without ever resorting to mark and sweep collection. The remaining difficulty, collection of cyclic structures, is believed not to be of importance to most practical applications. Comparison of performance characteristics with mark and sweep collection indicates that for most applications the strategies based on reference counters will be both simpler and more efficient. For applications requiring real-time or conversational interaction the advantages are even greater, with reference counter systems incurring only about a fraction of the penalty of mark and sweep systems when guarenteeing the list processor will not have to wait on the collector.

Further work is need to model various implementation strategies against a broad spectrum of applications. After selecting optimum or near optimum parameters, such as counter size and cache organization, the details of a hardware implementation could be effectively designed and implemented. The availability of a family of LISP processors based on such a strategy would greatly enhance the applicability of artificial intelligence for applications as diverse as weapons control, space station

monitoring, appliance control, and conversational interface.

REFERENCES

(1) Rich, Elaine, Artificial Intelligence, McGraw-Hill, New York, NY 1983.

(2) Winston, P. H. and Horn, B. K. P., LISP, Addison-Wesley, 1981.

(3) Cohen, Jacques, "Garbage Collection of Linked Data Structures," ACM Computing Surveys, Vol. 13, No. 2, Association for Computing Machinery, Sept. 1981, pp. 341-367.

(4) Greenblatt, et. al., "The LISP Machine," Artificial Intelligence: an MIT Perspective, Vol. 2, The MIT Press, Cambridge, MA, 1979, pp. 345-374.

(5) The LMI Lambda: Technical Summary, LISP Machine, Inc., Los Angeles, CA, 1984.

(6) Hickey, Tim and Cohen, Jacques, "Performance Analysis of On-the-Fly Garbage Collection," Communications of the ACM, Vol. 27, No. 11, Association for Computing Machinery, Nov. 1984, pp. 1143-1154.