Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful

by Jon L White
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 01239. July 8, 1980

ABSTRACT

The possibility of incredibly cheap, fantastically large media for storage gives rise to a realistic LISP memory management scheme under which GC may be postponed for days, or even indefinitely; the idea is encapsulated in the acronym "DDI" — "GC? Don't Do It!". Tertiary memory is used to archive pages of the LISP environment which are perhaps reclaimable, but which have not been proven so; whereas the standard technique of "paging" is used to swap active data from the main memory to a secondary store such as magnetic disk. Some scenarios are presented considering a variety of currently-available technologies, and of one speculative possibility — videodisc — by which a requisite compactifying GC would be done "overnight", or over the weekend. With enough tertiary available, one design could last for over 12 years without a GC. "Write-once" memories, probably unusable for most applications, would not be at a disadvantage here.

Table of Contents

---

1. Introduction and the Basic Problem of GC

The possibility of incredibly cheap, fantastically large media for tertiary storage gives rise to the question for LISP users "What is Garbage Collection done for, and what does the alternative cost"? On first- and second-generation machines, and in most mini/micro computers, the GC is *necessary due to exhaustion of the number of memory cells available*; typically the formerly used but now inaccessible cells were reclaimed by a "mark-and-sweep" GC. Later on, with the advent of virtual memory, the problem became one of exhaustion of the number of addresses available; also, there was the need to reduce the number of pages in the "working set" [/reference/ Fenichel and Yochelson; also Multics version of Maclisp addresses this issue]. Typically, these later systems used a "stop-and-copy" method, instead of a true collection of "garbage" cells, in order to compactify the working data. In a very large address-space machine, consideration must also be given to the requirement of the number of "core-resident" pages in the operating system necessary to support a large virtual address space [for this reason, many time-sharing installations have an administrative restraint on the actual size of a user's virtual memory; e.g., many VM/370 users find themselves limited to a 1Mby (= Megabyte) "machine", although a 16Mby "machine" is technically feasible].

Two problems arise of ocasionally insurmountable proportions: (1) An embarrassingly large time pause while the main process is locked out and the GC is running, and (2) a "thrashing" of the paging device while tracing or descending data-structures distributed somewhat randomly throughout the address-space/memory. In 1978, H. G. Baker presented a memory utilization strategy which overcame the first obstacle [/reference/ Baker], and it will be assumed that the reader is at least casually familiar with this "real time" scheme; although not immediately obvious, the second obstacle was still present to an annoying degree in this scheme [/reference/ Lieberman and Hewitt; also private communication from Greenblatt]. Herein we present a rationale for a memory utilization strategy which treats GC as "harmful", and avoids almost all stages of it. A kind of "Lipschitz condition" for memory utilization, desirable for programs which use temporary data, is introduced; it is shown how a novel use of a portion of the Baker GC algorithm maintains this "condition" in the virtualized machine- and LISP- layouts herein presented, and how a real GC would tend to destroy this condition. Programs running in one of these LISP systems will not be slowed down by the lack of GC, but when the GC becomes critically necessary, it will be a lengthy process — possibly requiring between ½ and 30 hours depending on the amount of accessible data, and may require short-term usage of a very large main memory. With such requirements, a GC would be

a "harmful" operation, due to the time involved and to the need for temporary use of an expensive component, namely the large main memory. Resource sharing of such a component, were it done on a once-a-week or once-a-month basis, could turn into a commercial enterprise, humorously suggested herein to be called "Rent-A-Mem, Inc.". Service might be by means of a mobile unit which attaches to the computer-utility plug/port on the exterior of an office building housing one or more large-scale LISP-supporting computers.

This strategy assumes only modest design changes in existing central-processors and time-sharing operating-systems, and indicates a place where an "outboard" micro-computer optionally placed in the memory system could be a tremendous help - named perhaps the "Intelligent Memory" by analogy to the Intelligent Terminal. "Secondary memory" is viewed as something like a currently available magnetic disk; "paging" onto secondary would be done automatically for LISP by a supporting operating system, using pretty much the existing hardware and software technology of "paging". "Tertiary memory" is to hold pages of the LISP address space which haven't been referenced "for a long time" — so long, in fact, that future references, if any, to data on these pages will probably be of low overall density, or nil. These pages, by virtue of being "old" in the sense of having not been referenced recently, will be called "archaic". The medium for tertiary may be any of several new technologies, such as videodisc, bubble domain, the IBM 3850 MSS, or even high-density serial magtapes. Happily, a write-once memory, such a videodisc may turn out to be, is as useful as any other for the tertiary memory required in this proposal. An interesting aspect about a serial tape is that one may just as well view it as a write-once meduim, not because it would be difficult to overwrite an existing record, but because of the sequential layout that the write-once condition imposes. Although the tape would be "slewed" back to read prior data, all writing would take place at the active end of the tape, so that most write-out requests would not have to wait for any "slewing"; it is the write-out for archival purposes which would be the most frequent operation on the tertiary memory. The value of migrating to tertiary, rather than the costly "harmful" alternative, is that by not re-cycling addresses, or at worst re-cycling them on the basis of once a week [or once a year!], the system is not required to *prove* that there are no chains of accessible references ultimately reaching that page. Most likely, there aren't any such chains — the "old" page is probably truly inaccessible — but to *prove* that is comparatively very costly, requiring that the GC *finish* its stubborn march through all paths originating from the roots.

Another proposal [/reference/ Lieberman and Hewitt] attempts to stratify "pages" according to a "generation" number, that is, according to the time at which consing first took place on them. Thereby, a page (or "chunk" of memory) with a given generation number may be reclaimed by a GC pass which ignores all data on chunks of lesser generation numbers. For example, if it has been determined to "GC" a particular group of chunks, and they are situated near the most recent 3% of constructed data, then the GC mark phase need trace through only about 3% of the data, instead of all 100%. The hope is that temporary storage will be quickly reclaimed, almost immediately after it has ceased to be useful, but there is still the open question of how best to determine which chunks to try to reclaim at any given point in time. Furthermore, it would be difficult to estimate the efficiency of the overall reclamation process without actually running it; the GC must still *prove* that there are no accessible data in a chunk to be reclaimed, but it is hoped that the incremental nature — returning recently-used temporary chunks quickly — will improve performance.

But, when in doubt, leave it out! or, as the idea in this paper will henceforth acronymically be called the "DDI" method, "Don't Do It". The consequences of not "Doing It" at all might at first appear to be the age-old problem of (1) having data distributed too thinly over too many pages, so that most data-references cause a page-exception, and (2) an outrageous amount of "garbage" is being dumped onto the secondary or tertiary memory system. The first problem can be rather simply corrected by using only part of the Baker GC algorithm [/reference/ Baker], the "transporting" part which copies objects from "oldspace" to "newspace" automatically upon reference to them; thus stuff concurrently being utilized will tend to clump together spatially. The second problem is handled by a minor design variant of hardware and software, which would be applicable to modern large-address-space computers like the Digital Equipment Company VAX, the S-1 which was designed at Stanford University and is being built at Lawrence Livermore Laboratory [/reference/ McWilliams et. al.], and even conceivably some of the 32-bit mini- and micro-computers which will continue to appear on the market.

Two terms will be used frequently in the remainder of this presentation, and need to be defined:

Q - quantum of LISP data, usually a "pointer" occupying one word. Certainly it has to be big enough to hold any address in the virtual memory configuration - if we expect a 2^{27}-bit address space of words, then a Q would have to be at least 27 bits long. "Q" should not be confused with "cons" cell, or "pair" cell — two Q's are stored into a "pair" cell, and under this memory configuration, two words [or Q's] are needed for "doing a CONS". The notation "Q" will be used when it is intended to stress the contents as LISP data, whose meaning must be "interpreted" by, or understood in the context of, the LISP system; the word "word" will be used in the more general computer sense. Either

one may be used when denoting information capacity.

Heap - a segment of pages where LISP consing takes place. In any LISP, it is desirable to distinguish several such areas, at least to the point of having a "static" heap. In "static" storage, which is discussed further below, constructed objects are expected to have some permenance; the normal GC neither "sweeps" them up, nor generally descends through them, except for specially designated "exit" vectors.

2. A Gigantic LISP Virtual Machine

We assume that the cpu will have a memory cache, with addressing to the byte, and a relatively full complement of instructions, of variable length as in the IBM370 or the VAX. Addressing to the byte rather than to the word is not wasting any limited resource (addresses), but rather facilitates character string operations and the efficiency of instruction storage. In order to facilitate the LISP memory management, there will have to be additional specialized instructions for access/update to LISP data, such as would be needed for the primitive LISP data. For the "pair" data type, CAR and CDR, are the basic selectors, RPLACA and RPLACD are the basic updators; for sequence-type data, like STRINGs and VECTORs, ELT could be the basic selector and SETELT the basic updator. These particular instructions are especially needed to insure that the "transportation" phase of the Baker GC algorithm is done; but even if these aren't primitive op-codes, and aren't easily microcode-able, then the LISP operations can be called as "out-of-line" subroutines, with some subsequent time slow-down. See section 2.3, "Transporting for Compactness ..." for a discussion as to why an object might have to be moved when referenced. For typical realizations of LISP — the kinds of primitive data types desired, and so on — see the "Standard LISP Report" [/reference/ Marti, Hearn, and Griss], and the two LISPs described by White [/reference/ White 1978, and White 1979].

Some extra paging registers shall be described below, which facilitate access to archaic pages, but these should not be critically necessary. However, for the gigantic machines used as examples, it will be quite important to permit an "astronomical hole" in the address space [where no regular page-table support is apparent] without unduly straining any system facility. User memory references to that area designated "archaic" will have to be trapped for service by the LISP sub-sytem.

It will be desirable to have the memory accessing mechanisms not only "cache" references, but also "go indirect" through "invisible" pointers found stored in memory cells [/reference/ Greenblatt]. "Invisible" pointers are to be installed in LISP storage objects whenever they are transported, for whatever reason. Even the lowly function "EQ" would have to have its arguments "dereferenced" — chasing down any invisible indirection chains — if they point to storage objects. Also, the prospect of reducing the initial storage for lists, by linearizing them at construction time [/reference/ Bobrow and Clark], makes it desireable to decrease the number of type code bits by two or three, and use those extra bits for "cdr-coding". Admitting "invisible pointer" codes, and "cdr-coding" into the architecture reduces by two or more the effective number of bits for data use in a Q; for the machines listed as examples in section 2.1, only those with fewer than 7 type code bits might feel this impact.

It would be nice to have a stand-alone, special purpose machine for LISP use, such as the one at Xerox PARC [/reference/ Deutsch] or the one built at MIT [/reference/ Greenblatt et al]; but any large-address, economical machine should be workable, in varying degrees [the VAX 11/780 may be a typical candidate]. The difference between one and another appears primarily in the size and flexibility of microcode available for the LISP part of the design, and the effect that lack of such microcode may have on LISP running speed. [The author, and several others at this Laboratory, conjecture that a memory cache may prove to be a more powerful facility than user-writeable micro-code — especially on a machine with a fairly supportive set of instructions and addressing modes]. There is, in fact, no need to be a "LISP machine", but such task could run under the supervision of a moderately friendly time-sharing monitor, along with a more pedestrian intermix of, say, FORTRAN and COBOL. The only special co-operation needed from the host operating system would be to ensure that: (1) each big, long-term LISP job has its own dedicated tertiary memory facility — which might just be a platter in a multiple disk layout; (2) accessibility for each page must be under control of the LISP job itself, and attempted references to non-existent pages, if not followed through the "archaic page bag" as described below in section 2.2, must return control to the LISP page-exception handler so that it may simulate the memory accessing needed for the "archaic page bag".

2.1 Address/Memory Sizes

We will assume, based on previous experience that 1MQ (= 1 Mega Q) of main memory and about 100Mby to 300Mby of disk would be at the heart of the memory system for a single-BIG-user system [private communicaton from Hewitt, and also from O'dell]. A "BIG" user is one that demands a very large working set of pages — a system may comprise thousands of utilities all in the same address space, even some as big as an interactive, intelligent editor, but still not become a BIG user. Many additional small users may be added to a single or multiple BIG-user system without much loss; but a BIG-user count of 10 would increase the need for

main and secondary memory, although not to a proportional extent [perhaps 4MQ main memory and 500Mby to 1000Mby of disk, depending on the raw speed of the processor]. The tertiary memory system needs to be big enough to back up the entire primary address space, for each simultaneous user, and a certain factor more for rewrites on a write-once medium. The IBM370, and many mini-computers [also MIT's LISP machine] have a span of 2^{24} bits, or about 16Mby [= 4MQ's], and this would fit handily on the smallest of modest disks; however, the VAX and the S-1 permit a user address span of about 2^{31}, or more than 2000Mby. Note that the VAX uses the high-order bit of its address space to mean "system" area, so that operating-system services may be entered with exactly the same instruction which enters user subroutines; this is why the 32-bit address space can support at most 2^{29} Q's of user data reference, but this crinkle on the addressing scheme does not impact usage of the other bits, nor does it affect the LISP design, and so it will be ingored. A reconfigured use of the VAX address to permit 5 type code bits would still demand 2^{27} Q's, or 512MQ of address space, and this appears to be smallest size for which one might want to consider "playing tertiary tricks".

We will also consider two hypothetical machines of gigantic proportions, the G-36 and the G-41, [presumably manufactured by the Gargantua Computer Company] which have a 48-bit word size, 12-bit byte, and respectively a 36-bit and 41-bit address space. The memory spans then are respectively 16GQ (= 16 Giga Q's) and 512GQ, which amounts to 24 Terabits! The smaller one could be serviced by a 64K-Megabyte tertiary system [Note: not, 64Mby, but about 65536Mby].

For illustrative purposes, we assume a page size, and disk track size, of 2048 Q's; this permits the calculation of page-table sizes correct to within a modest factor, except for the VAX, whose hardware has a much "finer" page size of 128 Q's [if the demands imposed by so small a size cannot be corrected in the hardware, then these page-table size estimates will be too low, for the VAX, by a factor of 16]

2.2 Memory Layout and Paging Registers

There needs to be four major "paging regions" in the main memory — accounting for four independent, randomly-accessible data segments — giving rise to three pairs and one set-of-three internal, per-process machine registers which delimit the ranges of these "paging regions". [The normal hardware layout of the VAX has two segments, called P0 and P1, but makes do with only two registers, instead of four, by fixing the low end of one area at virtual address 0, and the high end of the other at virtual address 2^{31}—1. We identify these "regions", with size estimates, as follows:

Page-tables and PDL	0.25MQ
Static heap	7.75MQ
Living heap	4.00MQ
Archaic page bag	1.00MQ

The sizes allocated to the individual regions are quite tentative, and except for avoiding extremes, it would be difficult to estimate accurately the effect of region size on system performance without actually doing a number of trial runs. It should be pointed out here, by contrast with other schemes, what these regions are not:

1) they are not spatial localizations of some coincidentally-related problem data, which the user [or "smart" program] assigns to its own area; this is for contrast with one aspect of the MIT LISP machine notion of "areas", which permits a "smart" programmer to construct spatially compact structures, in its own area, regardless of any other co-incidental consing or GC-transporting going on. [/reference/ Greenblatt, and also Bishop]
2) they are not areas containing objects all of the same data type, so that the address part of a Q contributes some small amount of information to the type code part [/reference/ Steele] except for the division between the living heap and the "static" area, they are not isolated areas, with few [or none at all] pointers out of one such area into another. An area so isolated can be garbage-collected essentially independently of the other areas, and thus support a kind of incremental GC scheme. [/reference/ Bishop, and also Greenblatt et. al.]

Capacities of Several Machines Compared

Machine name	word size in bits	Log2 of user address-space size in Q's	No. of type code bits for LISP	basic byte, or character size	No. GQ of tertiary needed
VAX	32	29	0	8	0.50
VAX*	32	27	5	8	0.13
S-1	36	29	5	9	0.50
IBM/370	32	22	8	8	none
G-36	48	34	12	12	16.
G-41	48	39	7	12	512.

4) They are not tables for managing a tertiary "back-up" to the system's secondary memory. In fact, both tertiary and secondary are direct "back-up"s to the main memory, but they are reached through different mechanisms — the one as a part of a standard "paging" scheme, and the other as part of a novel page archival scheme which substitutes hashing for the usual page-table/page-register support.

They are identified for the purposes of setting up independent paging resources, and are much more like "segments" in nature — the requested registers are in fact segment delimiters.

Let these four sets of registers be called

PDL-lo,	PDL-hi	boundaries for the page-table and PDL area
STA-lo,	STA-hi	boundaries for the Static area
LIV-lo,	LIV-hi	boundaries for the Living heap
TRS-lo		low boundary of non-automatic transport region
OLV-lo		the original value of LIV-lo
ARC-lo,	ARC-hi	boundaries for Archaic page bag

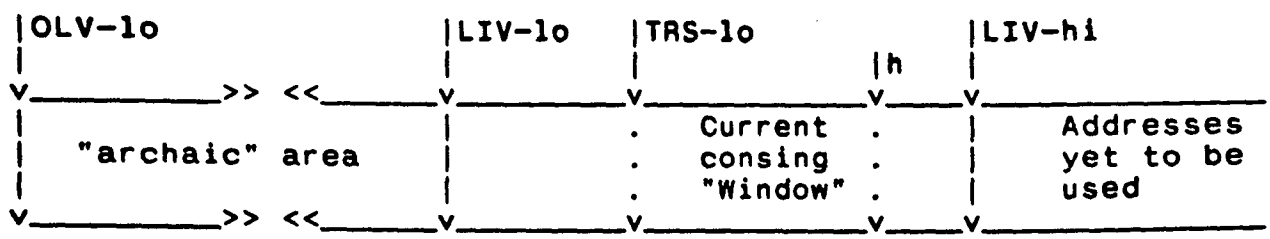
The TRS-lo register will normally split the Living heap — i.e. lie somewhere just above the address in LIV-lo — and any object stored at an address below TRS-lo, when referenced, will automatically be "transported" up into the consing area of the Living heap. Archaic pages start out as part of the living heap. As consing continues, the high end of the living heap is advanced upwards, and the low end of it is also advanced upwards so that its total length doesn't exceed some predetermined limit, say LIV-length; it may bloat up and thin itself down, like an accordion expanding and contracting, but it will never exceed that limit. LIV-length will depend upon many factors of system performance, and should be dynamically adjustable. But the pages which are advanced over by LIV-lo as it marches upwards are tagged as "archaic", and are immediately put into the archaic page bag, which is a ring-buffer of archaic pages. There will be no page tables entries for any of the archaic pages, in that part of the address space marked by the original LIV-lo and its current value; instead, if a memory reference is attempted to that area, then the page number of the desired address is used as a key to search the bag, [possibly with associative hardware help, or if necessary just hash coding]. For example, suppose the address {OLV-lo} + 3340012l[8] is being used, and it falls into the archaic area; then its page number, {OLV-lo} + 6700[8], is searched in the bag, and is found at, say, the 25'th entry; then that reference may be mapped into {{ARC-lo} + 25.} + 121[8]

The total length of the archaic page bag is fixed, and small by comparison with the "archaic" area; each slot represents where a page may be stored, before it is migrated, "oldest first" to tertiary memory. Associative registers should speed the "searching" of the keys to this region to find out if a particular page is currently there; there should also be an ordered index into the bag, probably a table of 1K words, maintained by insertion and deletion operations. Insertions/deletions to the bag will be rather infrequent, by comparison to other operations, but associative-lookup failures may occur relatively often, and it must be possible to search this index table to reload the associative registers quickly [or generate a exception trap if the search key is not in the table].

But what happens if an archaic reference is not found in the bag? Well, first note that pages dribble through the bag, and when reaching the end will be written out on tertiary memory; the bag need not be strictly a FIFO queue, but some attempt should be made to reorder it so that some "lesser-recently-used" page is the one picked for dumping out to tertiary. Thus it will always be possible to make room for one or more pages to be fetched back from tertiary, and reinserted in the top of the bag.

Page tables, whether "paged out" to secondary storage or whether resident in main memory, need to be allocated to cover only these four regions; in the case of G-36, this might mean that only 16MQ's of address space need to be covered, instead of 16GQ's. At one word in the page table for each page covered, and with a 2KQ page size, this would mean one page of page tables,

Picture of Living heap:



instead of 4096 pages of page tables for covering the whole space. For the VAX scenario in section 2.1, with 5 LISP type-code bits taken from its 32-bit address, there are only 64K pages total, and 32 pages of page-tables would suffice to handle the whole space. This seems small enough to suggest forgetting about the archaic page bag, and just providing enough page tables for every address in the space; most page tables, then, could be paged out on secondary, with some process register indicating which ones are really in memory at any given time [obviously, the page with the page-table for the page-table region could not be paged out].

So why not dispense with the "archaic page bag" in the G-36 case also? 16MQ of "paging regions" *when fully occupied*, needs 8192 pages of swapping area support, and adding another 8192 pages to hold the page tables for covering the entire address space would require additional secondary capacity equivalent to about a T300 disk unit; this actually is not totally unreasonable. But consider the G-41 case: still up to 8192 of pages are needed for swapping the "paging regions", but about a quarter million pages are needed for page-table pages, and this amounts to more than 3200 Mby — which is a lot of disk storage to dedicate *just for one user's page tables!*

Since the movement of data from areas below TRS-lo into the consing region may tend to degrade system performance, it is desirable to isolate data which is *not* temporary, and place it in a static area from which it will not normally be moved. For example, a large application system may consist of hundreds of subroutines along with many constant data expressions; all such code and data may be placed in the static heap, for indeed they will never be reclaimed as long as the that application system is alive. Some applications packages might even have parameter settings which automatically cause them to be constructed, or loaded, into the static area. On the other hand a user might explicitly request a particular package to be placed in static space, even when not done so automatically, because he knows that it will be of continuing utility during his encounter with the system (for example, some temporary debugging facility). If not directly moved to the static heap by the programmer, an object could be subject to some heuristically-induced migration — after such-and-such a number of Baker transportations into the living heap [from archaic pages], by automatic analysis it might be decided that it would be worthwhile to migrate the module to the static heap, based on the projected likelihood that it will continue to stay around and be used.

2.3 Transporting for Compactness: The Pseudo "Lipschitz" Condition

There are three major reasons why a stored object, once it has been constructed, will be moved to another memory location:

1) It might be moved to the static heap, not particularly for compactness, although it might be "cdr-coded" at that time, but due to expectations that it is a relatively permanent object.
2) To make RPLACD work in a "cdr-coding" environment, there must be the possibility of installing an "invisible" pointer in one cell, and moving the object formerly stored there to an un-coded cell; this motion, like 1) above, is not for compactness.
3) The transportation part of the Baker GC moves cells from "old space" to "new space", thereby achieving a compactness of some sort.

Not being too concerned about reclaiming address space, it is the results of transportation number 3) which make it realistic to run LISP with a "DDI" garbage collector. For, ignoring the rather trivial effect of transportations 1) and 2) on fast-moving, temporary storage, doing this third phase of movement, into the living heap, ensures a kind of pseudo "Lipschitz" condition relating memory occupancy and access. Suppose M_0 is the address in which an object is stored just after it has been referenced at time T_0, and similarly M_1 is its address just after a reference at time T_1 [$T_1 > T_0$]; then there is a constant ϵ dependent on the length of the living heap such that

$$M_1 - M_0 < \epsilon(T_1 - T_0)$$

Two aspects of the ordinary Baker GC tend to degrade the degree of spatial compactness of data accessed in a temporally-compact interval:

1) the need to transport all the substructures accessible from a cell which was just transported [say, from M_0 to M_1], and
2) the "daemon"-like action of the scavenger part of Baker GC, which is busy at work transporting any and every object accessible from the roots, regardless of when [if ever!] it will be referenced by the program again.

It has been noted on MIT's LISP machine that a large program which confines itself to a small set of working pages for some modest time interval will still be burdened by the demand for a large working set, due to the requirements of these two GC aspects; performance is remarkably better when these parts of the GC are turned off [private communication from Greenblatt]. Thus the "DDI" method keeps the successful part of Baker GC, transportation from the archaic area, or "old space", but omits the degrading part.

Bobrow and Clark [/reference/ Bobrow and Clark, P.282-284] pose an interesting hypothesis of a similar nature: if T_1 and T_0 are, rather than the times of access, the times of creation, then the probability that M_1 contains a pointer to M_0 is dependent only on T_1-T_0. In their paper, empirical evidence was presented which showed this to be fairly true for pointers in the cdr part of a cell, but the results were pessimistic for the car part. The investigation showed that "cdr-coding" will likely be highly applicable in the kinds of programs from which the empirical data were taken; it will reduce the number of pair cells needed to store a list — up to a 50% reduction — by eliminating cdr links [see also /reference/ Clark and Green]. In object-oriented programming, such as in SMALLTALK [/reference/ Ingalls], more data is kept in vector-like objects rather than in lists, and "cdr-coding" may be less useful. Nevertheless, the goal of this "interesting hypothesis" is to support an encoding scheme which reduces the number of bits used to store the car and cdr of a pair cell, and thus reduce memory requirements, which should simultaneously reduce the number of pages in the working set.

On the other hand, the goal the the pseudo "Lipschitz" condition is to reduce the size of working sets by a dynamic, and invisible to the user, re-shuffling of stored data; such is independent of the use of a "cdr-coding" scheme.

3. GC Once a Year: Enough?

Unfortunately, even a gigantic memory must be viewed as a finite resource:

1) Consing a "new cell" on the average every 10 microseconds, 12 hours a day, 6 days a week, would exhaust a 32-bit address space [of bytes] in something a little over a week. The VAX with 128MQ of space would last hardly half a working day at that rate, but G-41 could hold off for around 12 years!
2) Static space can only be "cleaned out" by a full GC, and it is possible to exhaust this space; also it is possible to get one's application programs spread too thinly over it, with lesser quality stuff in between.
3) Tertiary memory is not just a second-level image of primary; since it holds pages *which generally must be updated when referenced*, either to install an invisible pointer in the referenced cell, or to update one already there. In a write-once medium, this will necessitate moving the page into secondary memory, and eventually back out to tertiary, in a new place. Thus the consumption of a write-once memory is dependent not only on the number of "new cells" taken in the primary address space, but also on the number *and timings* of references to archaic but still accessible data. However, a bunch of references to data all on the same archaic page and all within the time interval during which the page is moving down the ring of the archaic page bag, would give rise to only one rewrite request.

Therefore, there needs to be a kind of gauge for Tertiary/Address/Static spaces, which would indicate approximately how long one could go before a real GC becomes mandatory; let it be called "T/AS". Just as an automobile's gas-tank gauge, which when reading ¼ full, indicates to the average 10^4 mile-per-year driver that he ought to go to a gas station and fill up some time before next Wednesday, so a computer's T/AS guage reading "low" would indicate to the 10^4 MQ-per-year lisp user that he ought to phone-up Rent-A-Mem and let the machine GC itself over the weekend, while he goes off for a well-earned two days in Bermuda.

It would be impossible to trace through a write-once memory while installing "invisible", gc-forwarding pointers; the running of a true GC, then, should be accompanied by enough real main memory to hold all the pages on which there is accessible data. Since the GC happens so infrequently, there certainly is no need for an installation, whether single-user or multiple, to have permanently affixed that much memory; and while almost all applications will have data which can in fact be compactified into a very few MQ's , it cannot be ascertained in advance whether that will require tracing through a few thousand pages, or a few million. The difference between these two prospects is staggering: because the first access to a cell, during the copy phase of GC, causes the installing of a gc-forwarding pointer, then there can be relatively very few such probes to tertiary which require a write-back [thereby consuming another tertiary page slot]. Likely, too, the dynamics of storage distribution are probably not favorable towards assuming that only a couple thousand pages are accessible.

We need a strategy for the hard case. Thus we suggest the ocasional sharing of a gigantic, main memory among several computer sites; enough memory to hold simultaneously all the pages which might be accessible. There are two ways to do this:

1) the hardware will have an easily-switched option on the memory bus such that a portable bank of main memory may be temporarily attached. Such memory would be shared among a group of such systems, and could be accounted financially on a fee-for-service basis [weekends and overnight would be more expensive!]. There might even arise an independent contractor, named say "Rent-A-Mem, Inc.", which could deliver a standard interface, certain speed memory on an hour's notice, in high-computer-density metropolitan areas. The spectre of a mobile truck, with a huge prehensile cable, plugging into a "computer service connection" on

the exterior of an office building, may be not only amusing, but someday a reality. Instead of filling the building with petroleum products, it will be "vacuum-cleaning" it's computers of unused bits.

2) At 1 megabaud transfer rate, the active pages in secondary memory could be "dumped" into the tertiary in something like ½ hour at most. Then the videodisc can be removed and carried to a regional service center that has the large resource. This alternative is especially attractive if the GC frequency can be held back to once a month, and if the turnaround period can be less than a few days. [consider the parallel with the shutdown of a nuclear reactor for it's ocasional clean-out; even the dangers are similar — with shipment via public highways, there is the danger of thievery and inadvertent radioactive leakage of nuclear waste, and correspondingly there is the danger of industrial espionage and inadvertent "information leaks".]

This proposal, to use fast primary memory for compactifying the tertiary medium, is an ironic counterpart to a similar proposal of nearly 17 years ago, which used secondary for compactifying the primary memory [/reference/ Minsky; and also Edwards]

For real-time applications, there is the possibility of a second computer used to take an incremental snapshot, fully valid at some instant in time, of the first computer's memory system. Call the first computer a "control-clerk" and the second one a garbage-man. Just after the snapshot is taken, the garbage-man then whirrs away, finishing the GC at some point in time, during which the control-clerk has continued to cons along. Then, providing only that the garbage-man can run faster than the control-clerk, either due to a faster cpu/memory or to a low duty cycle for the control-clerk, the garbage-man may update himself to account for the transactions having taken place in the interim. Finally, within the scope of just a few seconds, there is a replacement of the clerk by the garbage-man, and the system is running with a nearly-100% full T/AS tank.

There are some applications where a GC will probably never be worthwhile. Doyle's TMS system [/reference/ Doyle] builds a data-base, from which very little will ever be deleted, and as the computation progresses, each "reason" as to why something was done is added to the data-base. In this sort of application, the "DDI" method will most probably be the only reasonable strategy of storage management.

4. An Outboard LISP Memory Interface

A certain amount of memory management can take place in parallel with, and somewhat independent of, the main cpu actions. For example, a mini- or micro-computer resident in the memory system, synchronizing and communicating with the main cpu, could supervise

1) the age counts of pages on secondary, and oversee the migration to tertiary of older ones, as the secondary system begins to fill up. This task would require information about the various pages which are in the archaic page bags, and would place virtually no load on the cpu.
2) the updating of tables when a page is removed off the low end of a living region, and queued for entry into an archaic page bag.
3) the retrieval from tertiary, and its installation into an archaic page bag, of a referenced page which was in none of the various "regions". For this operation, the main process will be blocked [of course, this is not a problem in a time-sharing system].
4) the continuance of a "transporting", initiated as described in section 2.3. Transporting a single "pair" cell requires almost no time at all; but moving multiple-word objects, and installing the necessary invisible pointers could take a long time. The cpu can continue, after it receives the selected component which it asked for, while a parallel process in the memory system continues the transportation of the whole compound object. In a "cdr-coded" system, it may be advisable to maintain list compactness, or at least partially so, when transporting; so even a pair cell transportation might initiate a lengthy process.

Acknowledgements: The author would like to express thanks to Henry Lieberman and Jack Holloway for discussions leading to the ideas in this paper. Some of these ideas are being tried out in NIL, a NewImplementationofLisp, and a debt of gratitude is owed to Professor Joel Moses and the NIL group at MIT's Laboratory for Computer Science for patience during this development [/reference/ White 1979]. Rich Zippel suggested the notion of a "LISP Memory Interface"; John Kulp and Tom Knight pointed the author to the videodisc technology; and L. Peter Deutsch provided helpful comments based on his experience with the Alto mini-computer. A special thanks goes to Professor Moses for his proofreading and critical commentary on the initial version of this paper.

Private Communicaton References

Greenblatt - Computer mail dated 27 AUG 1979 and 18 MAR 1980 regarding performance of the MACSYMA on the LISP machine, under paging and GC load.

Hewitt - MIT's LISP machine project started out with the assumption that 128KQ of main memory would be satisfactory, but that estimate has been increased to around 200K as of 1980; several of the LISP machines have already been augmented to 256KQ. Carl Hewitt, noting the rather low prices of main memory in comparison to other system components, expects to have a LISP machine with 1MQ attached.

O'dell - Jim O'dell comments, in some computer mail dated 3/17/80, on trying to run MACSYMA on one of MIT's LISP machines with 128KQ of main memory. The slow-down due to paging is "unacceptable" — 5 times slower than running on a time-shared KL-10 — whereas 256KQ was "a totally new ball game".

Bibliography

Baker, H.G., Jr.; "List Processing in Real Time on a Serial Computer"; *Comm. ACM 21*, 4 (April 1978), Pp. 280-294.

Bishop, P. B.; Garbage Collection in a Very Large Address Space; Technical Report TR-178, MIT Laboratory for Computer Science.

Bobrow, D.G., and Clark, D.W.; "Compact Encodings of List Structure"; *ACM Trans. on Prog. Lang. and Systems 1*, 2 (Oct 1979), Pp. 266-286.

Clark, D.W., and Green, C.C.; "An Empirical Study of List Structure In LISP"; *Comm. ACM 20*, 2 (Feb 1977), Pp. 78-87.

Deutsch, L.P.; "Experience With a Microprogrammed Interlisp System"; *Proc. 11th Annual Microprogramming Workshop*, Asilomar, Pacific Grove, CA. (Nov 1978)

Doyle, J.; "A Truth Maintenance System"; *Artificial Intelligence 12*, 3 (Nov 1979), Pp. 231-272.

Edwards, Daniel J.; "Secondary Storage in LISP"; A.I. Memo 63, M.I.T. Artificial Intelligence Lab, Cambridge MA (Dec 1963).

Fenichel, R.R., and Yochelson, J.C.; "A LISP Garbage-Collector for Virtual- Memory Computer Systems"; *Comm. ACM 12*, 11 (Nov. 1969), Pp. 611-612.

Greenblatt, R.; "The LISP Machine"; Working Paper No. 79, M.I.T. Artificial Intelligence Lab, Cambridge MA (Nov 1974).

Greenblatt, R., et. al.; "LISP Machine Progress Report"; A.I. Memo 444, M.I.T. Artificial Intelligence Lab, Cambridge MA (Aug 1977).

Ingalls, D. H. "The SMALLTALK Programming System Design and Implementation"; *Fifth Annual Symposium on Principles of Programming Languages*, 1978.

Lieberman, Henry, and Hewitt, Carl; "A Real Time Garbage Collector That Can Recover Temporary Storage Quickly"; A.I. Memo 569, M.I.T. Artificial Intelligence Lab, Cambridge MA (Apr 1980); (also submitted for publication)

Marti, J., Hearn, A., Griss, M., and Griss, C.; Standard LISP Report; UCP-60, University of Utah (Jan 1978).

McWilliams, T.M., Widdoes, L.C., Jr., and Wood, L.L.; The S-1 Project; Lawrence Livermore Laboratory, Livermore CA (Sep 1977)

Minsky, Marvin L.; "A LISP Garbage Collector Using Serial Secondary Storage"; A.I. Memo 58, M.I.T. Artificial Intelligence Lab, Cambridge MA (Dec 1963) (out of print).

Nadan, J.S. "Optical Information Storage and Retrieval Systems"; in Archival Memory Technology, Proc. of a Workshop Held at Carnegie-Mellon Univ. (Sep 28, 1978), Pp. 28-30.

Steele, G.L. Jr.; "Data Representations in PDP-10 Maclisp", A.I. Memo 420, M.I.T. Artificial Intelligence Lab, Cambridge MA (Sep 1977).

White, JonL; "NIL — A Perspective"; in *Proc. of 1979 MACSYMA Users Conference* (June 1979), Pp. 190-199.

White, JonL; "LISP/370: A Short Technical Description of the Implementation"; *SIGSAM Bull. 12*, 4 (Nov 1978), Pp. 23-27.