Erik Sandewa

# A PAGED LISP USING THE DYNAMIC RELOCATION HARDWARE OF AN IBM 360/67

A PAGED LISP USING THE DYNAMIC RELOCATION
HARDWARE OF AN IBM 360/67

Robert I. Berns Stanford University

The language LISP presents a unique challenge for the application of techniques which have been developed for use in a paging environment. The nature of LISP with respect to accessing available space and the use of a garbage collector will be described in relation to a system which provides swapping at a hardware level. The major emphasis will be on the algorithms used to most efficiently execute in this environment, advantages gained in having hardware paging, and some statistics on the speed and efficiency of the system.

## I. Introduction

The Stanford Computation Center operates an IBM 360/67 with multiple partitions which include both batch and Time-Sharing [illegible]. In the batch partition there is a LISP system with 24K of free cell storage (IBM 360 double words) and 24K words for compiled code. Under other configurations these limits may be raised as high as 60K and 60K respectively. Using only the interpreter it is possible to make available to the user 90K of free cell storage. For most LISP problems solved locally these sizes have been found to be quite sufficient.

A Time-Shared version of the LISP system has been written which operates under a monitor that uses the dynamic relocation hardware of the IBM 360/67. This version was originally written to use only the physical memory provided for by the monitor. Under this restriction, the LISP system can only access 15K of free cells with no room for the compiler. To make it possible to run larger jobs such as those possible in the batch, a new version of the Time-Shared LISP was written. This version, using services provided for by the monitor, has the ability to use up to 64 pages (4096 bytes per page) of drum space for virtual memory. The total physical core available is 24 pages.

The monitor allows the LISP system to manage its swapping to some extent. In actuality, all control of page swapping is done by the monitor, and mearly provides the system writer with the ability to define an algorithm for managing his pages.

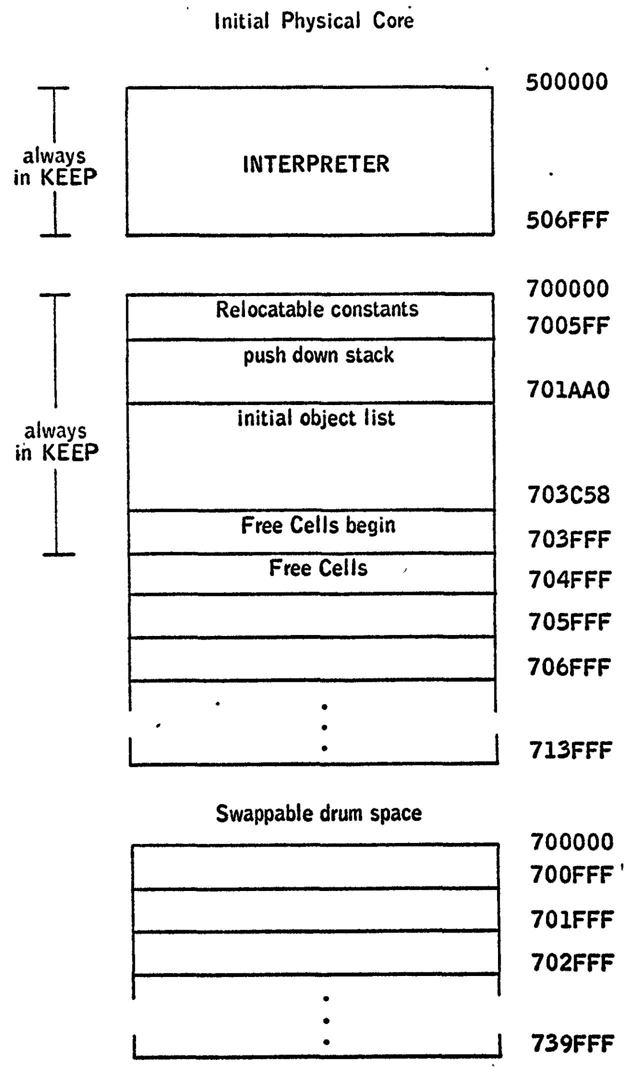
To do the work described above, it was necessary to do a major re-write of the garbage collector and the CONS routine as well as adding a new routine to handle the swapping. Some of the methods described are based partly on modifications of the work of Bobrow and Murphy (3) with respect to the CONSing algorithm. This paper describes additional algorithms to select the page which should next leave physical core, and some special considerations for the garbage collector.

In what follows I will be continually referring to the terms "KEEP" and "HOLD". KEEP may be thought of as the process of having a page of virtual memory currently residing on the drum brought into physical core. The responsibility for modifying relative addressing is shared between the hardware of the 360/67 and the Time-Sharing Monitor. This work is not done by the LISP system. "HOLD" may be thought of as the reverse process of writing a physical page out onto the drum and making the physical page available. Again, the responsibility of addressing is outside of the LISP system. Hence, a swap is HOLDing one page and KEEPing another.

## 2. Virtual Memory Allocation

Figure 1 shows a schematic of time shared LISP/360. This version is described without the compiler included. The addition of the compiler will increase the size of the interpreter slightly and will remove some pages from the total free-cell space for the collection of compiled code. However, the fundamental swapping algorithm will remain intact. As is true in BBN LISP the compiler and interpreter may interact with one another with compatibility over S-expressions and function definitions.

Figure 1
System Schematic
of Virtual Memory



The addresses shown are hexadecimal. Notice that the address for the system code shows it to be in a different place than the relocatable data. This is to aid the monitor in sharing the same copy of the subsystem among many users. The initial object list contains all atoms pre-defined by the LISP system (character objects, function names etc.). The relocatable constants, push down stack, initial OBLIST, and the first page of free cells remain in virtual memory at all times. This decision was based on the high usage of these items throughout a run. However, the garbage collector must still collect over all items starting from the initial OBLIST. The rest of virtual memory may be used for swapping.

## 3. Free Cell List Maintanence

Initially , the user is given seven pages of free-cells, the first two being the initial object list. A separate free list pointer is maintained for each available page. When a page boundary is crossed the old pointer is saved and the one for the new page becomes active. This is to aid in concentrating the creation of lists on single pages. Further work on doing this is handled by the CONSing algorithm. Having this compactness (or linearity) is a desirable property, for if lists can be kept from crossing page boundarys too frequently, then scanning these lists will cause fewer swaps.

As new space is required, and the current page has no more space, a check is made of the free list table. If a page exists with space it is used (again the CONSing algorithm will try to get a particular page to increase compactness). When no space is available on any active page a garbage collection takes place. If the garbage collector decides more space is needed than the amount collected a new entry is made in the free list table and a new page becomes active and is initialized. Initialization is the process of linking each double word to the following one and placing a special mark in the last cell indicating the end of space on that page.

## 4. The Garbage Collector

The garbage collector must collect on all active pages whether they are in KEEP or HOLD (the swapping algorithm is used to bring in HOLD pages as they are needed). It is after the garbage collection takes place that the decision is made whether a new page should be added to the list of active pages or not.

The criterion used is that a new page is added if the number of cells collected is less than 10 percent of the total number of current free-cells. The choice of 10 percent is arbitrary and seems to work well for most test cases. Two possibilities for change exist here. Have the percentage be a parameter set by the user, and/or determine the percentage dynamically based on the current usage. Ideally, it would be best for the larger programs to get pages rapidly at the beginning (a high percentage) and then less as the program's demands begin to level off.

When the decision to add a new page is made a number of possibilities exist. There may still be room in virtual memory to add a new page without swapping an old one; if so, this is done. However, when no KEEP space is available some page must be put into HOLD so the page may be initialized and hence the swapping algorithm is again used.

The swapping algorithm itself makes decisions as to which page should be swapped out based on criteria to be discussed. However, the swapping done in a garbage collection should not affect these criteria as they are based on the status of the running program and not the incidental garbage collections. Hence, tables used in the swapping algorithm are saved upon entry to the garbage collector and restored upon exit.

## 5. Algorithms

SWAPPING OF PAGES. When a referenced page is not in KEEP status some decision is necessary as to which page should leave core to make room for the referenced page. The obvious choice is the page which has been referenced the least. However, to count the number of references to a page or the length of time that has elapsed since a page has been referenced requires either some special hardware or some intricate software simulation of that hardware. COHEN (4) describes some of these techniques in his paper. In it, he concludes that time of inactivity is one of the best criteria for choice of the leaving page.

I propose a method for choosing a "reasonable" page to leave without computing an unreasonable number of statistics. Consider the following two factors:

1) Page references tend to "clump". Particularly as the result of linearity. So it is preferable to keep in physical core "new" pages and throw out "old" ones.

2) Some pages are heavily referenced, regardless of age, and will be required almost immediately if they are thrown out.

The first factor can simply be measured by the number of pages which have been brought in (number of page interupts) since the page under consideration. The second factor can be based on historical data, namely how many page interupts on the average have occured between swap out and the next swap in of the page. The relative weights given these factors determine the "conservativeness" of the algorithm, i.e. how long a page is kept in physical core simply on the basis of (perhaps) ancient swap data. If physical memory is small, it is preferable to be able to use all pages of physical core for swapping as opposed to keeping a page because of high activity, so factor 1 should dominate. With large physical memory, some "deadwood" can be tolerated for better average swapping behavior.

With this in mind the following was done. An in-core table with three entries for each available page (in core or on the drum) is created with the following information contained in it:

1) SC - The Swap Count - The number of times the page has been swapped out.

2) SD - The Swap Differential - The total number of swaps which have occured while the page was not in Physical core.

3) M - The Maximizing statistic - This is described below. The page with the largest M is the page to leave core.

Other variables used below in describing the algorithm are:

4) CCI - The Current Count In - The total number of swaps which have taken place when the page was last brought into physical core.

5) S - Swaps - The total number of swaps which have taken place.

6) K - A constant of proportionality.

The value of SD/SC is then the average number of times a page has been out of core. Hence, the page with the maximum SD/SC would be the page to throw out. However, a page could get a very small SD/SC by having an early high activity and never be swapped out. To compensate for this we may add to this figure the number of times the page is in core. This would be the total number of swaps which have occured (S) less the count when the page was brought in (CCI). So the longer the page is in core the greater (S-CCI) will be. Eventually, this value will become large enough to dominate the following expression:

SD/SC+K*(S-CCI)

The page with the largest value for this expression is the page to leave physical core. The constant K is used as the weighting factor described in the previous paragraph. For our system a K of 4 was chosen. The computation of S, SC, and CCI are obvious by definition, SD may be computed by subtracting S from the current SD when the page is taken out, and adding S when the page is brought in. From an examination of the equation we can see that S and K are constants for each page per swap and may be eliminated from the computation. Also, the value obtained is constant while a page is in keep and hence we may compute it once when a page is brought into KEEP and the value stored in the table. The algorithm becomes therefore:

Compute the maximum value of M for those pages which are in KEEP. M is computed from M=SD/SC-4*CCI each time a page is brought into KEEP.

This table can serve additionally to give the status of the because M containing a non-zero value means the page is in KEEP, M of zero can be used to mean the page is in HOLD, and an SC which is negative can be the initialization to show a page which has never been referenced.

CONSING ALGORITHM. Since CONS is the only LISP function which requires getting new space, it is at this point that the decision as to what page to get the space from should be made. We want to keep the lists which are created linear and on as few pages as possible so that references to lists do not cause an excessive amount of swapping. Therefore, CONS should try to get new space from the same page that the items it is CONSing are on. Preference should go to the second argument as most lists are scanned by moving along the CDR elements. The method above is described by BOBROW and MURPHY in their paper. Their method is slightly more sophisticated due to some additional information about the nature of ATOMS. For the system being described here all that is done is the following:

To construct Z=CONS(X,Y), if there is room on Y's page put Z there, else if there is room on X's page put it there, else choose a page with space on it that is in core. Z is placed on X's or Y's page if there is room regardless of whether the page is in physical core or not. This is because the number of times we scan this list will usually be considerable as compared to single swap necessary to get X's or Y's page.

## 6. Discussion

In examining some test cases, for which statistics are given below, it was found that a major problem is in the object list. The scan for a particular item requires a linear search of OBLIST and the comparison requires that each page on which an item of the list occurs be brought into core, resulting in much additional swapping. A hashing method for storing objects could considerably reduce the number of swaps. The system being discussed in this paper is a modified version of the BATCH LISP being used at Stanford and under normal conditions it was found that the changing over to a hashing scheme for the object list did not increase the efficiency of the batch system. It was decided that to do this for the timeshared system would involve too much of a program re-write, and so we decided to put up with the OBLIST problem until another method for solving it could be employed. As the OBLIST is scanned only during the initial reading in of functions where paging activity due to other sources is usually relatively low, this problem has not created a serious threat to the overall efficiency of the system.

STATISTICS. The test cases used to gain some representative statistics were the Expression Recognition Routine (ERR) described by ROSEN (5); the METEOR language written in LISP described in Information International Inc.'s book (6); a polynomial simplifaction program; and some highly interactive student problems. The polynomial simplification program was used because of its size (approximately equivalent to a card deck of 800 cards). This was to check the effect the object list problem would have. METEOR and ERR tend to create rather long lists and hence help in checking how well the algorithms linearize lists. The student problems are primarily exersises in defining and testing functions so that much interaction was required and response time would be more observable.

One important fact should be mentioned at this point, and that is with reguard to "number of swaps". While I have talked about moving a page from physical memory (KEEP) to the drum (HOLD) as constituting a swap, there may indeed be no drum reference made. This may be true for one of two reasons. First, if the page being put in HOLD has not been changed then no drum reference to store the page is made. Secondly, the monitor, which may at one time be handling many sub-processors may find that even though a request for putting a page in HOLD is made, there is sufficient core available to avoid making the drum reference at that time (which means essentially that the size of physical memory can vary). Since frequently in LISP large lists are scanned but not changed, it is quite plausible that the actual number of drum references may therefore be somewhat less than the number of "logical swaps". Indeed, under controlled conditions this seemed to be true for a considerable number of test cases. For the test cases mentioned above the number of swaps was about 30 percent greater then the actual number of drum references. For other tests this figure varied from 15 to 40 percent.

To test the efficency of the system counters were put in to count the number of CAR's, CDR's, CONS's, swaps, garbage collections, and swaps done in the garbage collector. Additionally, time of run, time spent in swapping, and time spent in garbage collection were recorded. As is mentioned in (3) it would be expected that if storage were distributed randomly then the percentage of cell references which caused swaps would be about the same as the ratio of amount of physical space available for the program to the amount of virtual space used. It should be pointed out that the number of CARS and CDRS is actually greater than the number shown in the statistics. This is because the LISP system does its internal CARS and CDRS "in line". The number shown are those called for by the user program. To get a true relationship between CARS, CDRS, and swaps the swaps counted are also only those swaps encountered during user calls to CAR and CDR.

A test of the linearity of lists can be made by checking how many swaps were necessary during a garbage collection. The more linear the lists the fewer swaps necessary. Since each active page must be referenced, the minimum would be one swap for each page that could not reside in KEEP space.

Figure 2 describes the values for the above mentioned factors. The size of physical memory for these runs was 10 pages. Cases 1,2, and 3 represent the ERR and METEOR programs run together, the polynomial simplification program, and the student problems respectively. Lines G. and H. show that the paging algorithm seems to be rather efficient for all three cases. For cases 1 and 3 it also appears that the lists formed in the system are rather nicely linear, as shown in I. for the reason stated above.

The problem in case 2 demonstrates that the program is rather large and most activity is during the reading in of functions where the OBLIST problem mentioned above takes its toll.

Though the statistics show that the algorithms used are quite effective, it is the response time for the user which actually determines the systems usefullness. For small programs and larger ones with much interaction, response time is good. For larger, compute bound jobs, response time is too long to make the system effective. This problem is due to the complexity of the Stanford system and its requirement of serving many diverse users in a multi-programming environment. It is felt that a LISP system such as the one described above would work remarkably well in general on a machine with a much more single-minded purpose.

Figure 2

Statistics

| | case 1 ERR-METEOR | case 2 POLY. SIMP. | case 3 STUDENTS |
|---|---|---|---|
| A. number of CAR-CDR-CON's | 172,128 | 141,268 | 31,398 |
| B. CAR-CDR-CON's page interrupts | 1,560 | 3,239 | 191 |
| C. number of garbage Collections | 131 | 81 | 22 |
| D. Garbage Collection page interrupts | 4,597 | 27,507 | 484 |
| E. Number of pages used (512 cells per page) | 19 | 35 | 20 |
| F. Average number of cells collected | 1,400 | 1,800 | 1,700 |
| G. Percentage of CAR-CDR-CON'S which caused page interrupts | 1.01 | 2.29 | 0.6 |
| H. Expected value of G, assuming random cell distribution | 47.37 | 71.43 | 50.00 |
| I. Average number of interrupts per garbage collection | 35 | 339 | 22 |
| J. Total time spent handling interrupts (seconds) | 67 | 377 | 18 |
| K. Total run time (seconds) | 272 | 1,191 | 130 |