MEMORY MANAGEMENT IN THE SMALLTALK-80 SYSTEM

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Science in Computer Science

in the

University of Canterbury

by

M. Ellims

University of Canterbury

1987

The goal of Computer Science is to build something that will last at least until we've finished building it.

fortune file, 4.3 BSD Unix.

Acknowledgements

There were many people who helped in some way in the preparation of this work, in particular I would like to thank Dr. W. Kreutzer for accepting me as a research student. I would also like to thank my supervisor, Dr. B.J. McKenzie for the help that gave and for reading the first drafts of the work; despite my novel spelling.

I would also like to thank Dr. M. A. McLean for doing the penultimate review of this work, and Professor J. P. Penny for his comments on Section 7.3.

Special thanks must go to Mr. P. D. Anderson for writing the interpreter for the Smalltalk-80 system at great personal cost.

I would also like to thank Robert Biddle for much valuable advice and help. I would also like to thank Linda Hicks for help in correcting the final draft of this work.

Abstract

This work presents an examination of the memory management area of the Smalltalk-80 system. Two implementations of this system were completed. The first system used virtual memory managed in an object oriented manner, the performance and related factors of this system is examined in detail. The second system implemented was based wholly in RAM and was used to examine in detail the factors that affected the performance of the system.

Two areas of the RAM based system are examined in detail. The first of these is the logical manner in which the memory of the system is structured and its effects on the performance of the system. The second field is the way in which object reference counts are decremented. This second field has potentially the large effect on the system's running speed.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter I

Introduction

Smalltalk-80 is the first public release of a system whose development was started in the early 1970's at the Xerox Palo Alto Research Center. The system has been designed and built around three basic concepts. These concepts are:

- a graphic user interface
- a "modeless" system
- object oriented programming

The user interface is perhaps the area where Smalltalk-80 has had the greatest effect on the computing community. The concepts that it developed have become the central paradigm of machines such as Apple's Macintosh computer. The user interface is built around three major blocks:

-windows

- menus, either alphanumeric or graphical

-the use of a pointing device to control both of the above features

The first two of these features are based on the use of a large bit mapped display. By using this as the display medium a large degree of control can be exerted on factors such as the size and positioning of windows, fonts used for text, the use of pictures for displaying information and the use of menus.

The pointing device (usually a "mouse") controls the display features of the system. The mouse is used to position a cursor on the screen. By using a combination of the mouse's position and three control buttons (or keys), the mouse can be used to reposition and resize windows, select and manipulate text and graphics and "bring up" menus onto the screen.

The second of the basic concepts is modeless programming. What this means is that there are no "modes" that a programmer or user of the system must enter into to work. That is there is no separate editing mode, command mode or compiling mode. What has been attempted is to build a uniform system that consists only of programs, each program having equal status with all other programs.

The language in which the Smalltalk-80 system is implemented is Smalltalk-80. This language is based on the paradigm of object oriented programming. In the system data structures such as arrays and lists are considered to be *objects*. Associated with each class of these objects is a set of *methods*. To perform an operation on an object it is sent a *message*. These messages are associated with the methods, for example if the following message was sent to an object that is a linked list *named* FirstList

```
FirstList addNewNode: 15
```

The message is "addNewNode". This invokes the method associated with this class and adds a new node containing the value 15 to the list "FirstList".

The textual form of a Smalltalk-80 program is compiled to a form known as bytecodes. Each bytecode corresponds to a *primitive method* or operation that is implemented as part of the lowest level or *kernel* of the system.

The lowest level of the system is of course the hardware. However just above this is the Smalltalk-80 *virtual machine*. This is the system's kernel.

The virtual machine consists of two parts; at the lowest level is the *memory manager*. This section of code is responsible for maintaining the integrity of the Smalltalk-80 memory space and for controlling <u>all</u> access to the objects of the system.

The second part of the virtual machine is the *bytecode interpreter*. This section of code drives the Smalltalk-80 system in a usually infinite loop. In each loop it selects a bytecode from an object and performs the operation or operations that it specifies. At the end of that operation the next bytecode, which was determined by the actions of the last, is selected and executed. It can be seen that Smalltalk-80 bytecodes have the same standing as instructions, but at a slightly higher level.

The task of the implementor of a Smalltalk-80 system is to implement the virtual machine. This thesis covers the memory management section of the virtual machine.

Chapter 2 gives a formal introduction and description of the system as defined in Goldberg and Robson [1982a]

Chapter 3 covers the decisions that were made about the system before the implementation phase was begun.

Chapter 4 gives the results of the initial virtual memory systems that were implemented and discusses the various methods that were examined for moving objects from RAM segments to disk segments.

Chapter 5 is used to present the results of a number of simulations of memory organisation strategies that were examined.

Chapter 6 presents the work that was done with the problem of reference counting and the large overheads that it held for the system. A number of systems are examined, both from the literature and methods developed here.

In chapter 7 various results that were obtained over the course of this work are presented. A possible direction for future work is also suggested.

Chapter 8 is a summary of the work that was done and the time frame in which it was accomplished. It also highlights certain factors affecting the performance of the system.

Chapter II

The Smalltalk-80 Memory Definition

2.0 Introduction.

This chapter gives a brief overview of the Smalltalk-80 memory management system as described in Goldberg and Robson [1982a]. This book describes the ideal model of the low level code that is used in the implementation of a Smalltalk-80 system. This chapter is included because comparisons will be made between the system as detailed here and the system at various stages of the implementation. The specialised terminology that is used in describing Smalltalk-80 systems is also introduced.

2.1 The System.

The Smalltalk-80 system consists of the hardware, a memory management section, a bytecode interpreter and the Virtual Image of the system supplied by Xerox. Each of the components is defined as interacting only with the level directly above or below it's level. Thus the only interaction between the interpreter and the hardware is via the memory management component of the system. The function of the memory manager is to allocate and deallocate memory and to do the bookkeeping for the system. The interpreter is used to run the actual Smalltalk-80 system which exists as a series of objects, termed the Virtual Image, whose interactions define the behaviour of the system.

2.2 Objects.

The Smalltalk-80 system is an object oriented system, that is all units that the system performs manipulations on have a well defined set of boundaries and properties. An object is a collection of values (of either 8 or 16 bits) arranged in a logically contiguous manner in the system's address space.

An object can be one of two types. The first object type is a contiguous array of 16 bit words that can be accessed either as words or as bytes by an offset from the object's start address. The second type of object is termed a small integer and this object occupies only 1 word.

Each object of the first type contains at least two fields of word size (16 bits). These fields are 1) the object's size in words and 2) the class of the object. The object's class will usually determine the meaning and format of the remainder of the data in an object.

Every object in the system is accessed via a pointer. An object pointer is a 16 bit word one bit of the 16 being used to define the type of pointer. If the type bit is set then the object is a small integer and the remaining 15 bits form the data part of the object. If this bit is not set then the word is an index to a table that contains the actual location in memory of the object. The type bit is defined to be bit 0 in Goldberg and Robson (see figure 2.1).

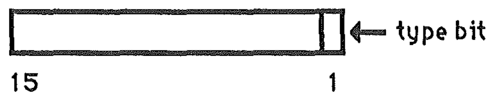

Figure 2.1: An object pointer as defined in Goldberg and Robson.

Objects of this type can be further divided on the value of their pointer bit. This usually defines the object to be either of a kind that:

- contains pointers to other objects, or
- contains other data such as small integers real numbers etc. The exception to this rule are objects of class METHOD which contain data of both types. The third word of this object class defining the exact layout of the data within the object it at run time.

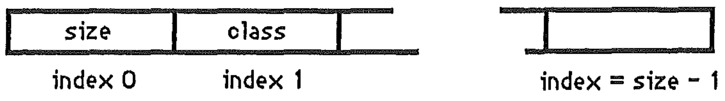

Figure 2.2: Layout of a non small integer object.

The second type of object is the small integer. This is a 15 bit signed integer that occupies one word. Thus it can represent integer numbers in the range -16384 to 16383 in twos complement form.

The effect of this is that even numbered pointers refer to "complete" objects (in that they exist elsewhere in memory) and odd numbered pointers exist as objects on their own right and have no separate memory allocated and no entry in the system tables. Thus small integers do not require storage except as an instance of a value.

We will use the term "object" here to refer to an object which requires storage in addition to its pointer value. A small 15 bit signed integer object will be termed a small integer.

2.3 The Address Space.

The definition of Smalltalk-80 memory management uses a segmented address space, where each Smalltalk-80 segment in memory has a fixed size of 64K words (128K bytes) indexed from 0 to 65535.

Objects are allocated space wholly within one Smalltalk-80 segment. This is because the memory manager deals only with units whose maximum size is 64K words. Objects that are allocated space within one Smalltalk-80 segment are the logical equivalent of a "segment" in systems such as the Burroughs B6700 series of machines [Organik 1973], where a "segment" maps onto a section of code such as a procedure or function or onto an an area of storage for a structure such as an array. Each Smalltalk-80 object can be viewed as a block of memory with a definite function; thus objects are in many ways the exact analogues of "segments".

The Smalltalk-80 segments can also be viewed as the equivalent of pages in a paged system, that is they delimit *physical* areas of memory and not *logical* areas in the way that objects do.

A complete address of an object in the Smalltalk-80 system comprises:

- its segment number and
- an index to the first word of that object (its size field) in that segment.

2.4 The Resident Object Table

Object pointers are indexes to the resident object table (ROT) which holds all the basic accounting data of the system. This information consists of the following:

CountBits :	8 bits used for reference counting on objects.
Obit :	1 bit, set if and only if the object uses an odd number of bytes in an object ie. the last byte of an object does not contain valid information.
Pbit :	1 bit, set if and only if the object contains pointers to other objects.
Fbit :	1bit, set if and only if the object points to an object that is not in use and may be reallocated.
U Bit :	Un-used.
SegBits :	4 bits, used to hold the number of the segment where the object's body is allocated.
LocBits :	16 bits, the location bits are the word index in a segment of the first word of an object.

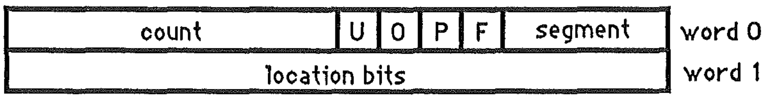

Figure 2.3: The 32 bits of a ROT entry as per Goldberg and Robson.

The combination of the segment number and the location bits provides the complete reference to an object.

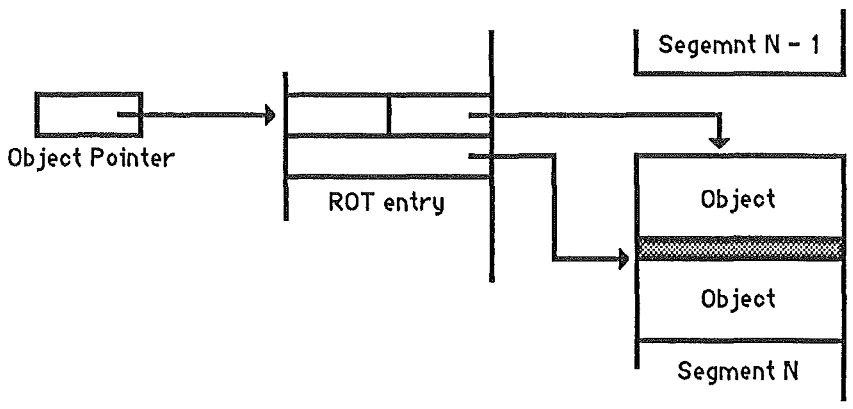

Figure 2.4: Object addressing.

2.5 Objects In Memory.

All of the Smalltalk-80 memory is contained in objects; these objects are either allocated or free. If an object is free it is then stored on a linked list of free objects. Several of these free lists exist, one list each for objects of size 2 through 10 words and one for larger objects. In addition, these 10 free lists are maintained for each segment in the memory address space.

The links of the list of free *objects* are formed by using the class field (Class Bits) of an object to contain the object pointer to the next object in that list. The class bits of the last object in the chain contain a small integer which cannot be an object pointer, thus marking the list's end. The first link of the list is referred to by an object pointer that is held in a "register". In a segmented memory system free objects do not need to be connected in the free lists via the ROT, as each 16 bit class field is wide enough to hold the address in a segment of the next free object. However the possibility of using this form of structure was not looked into.

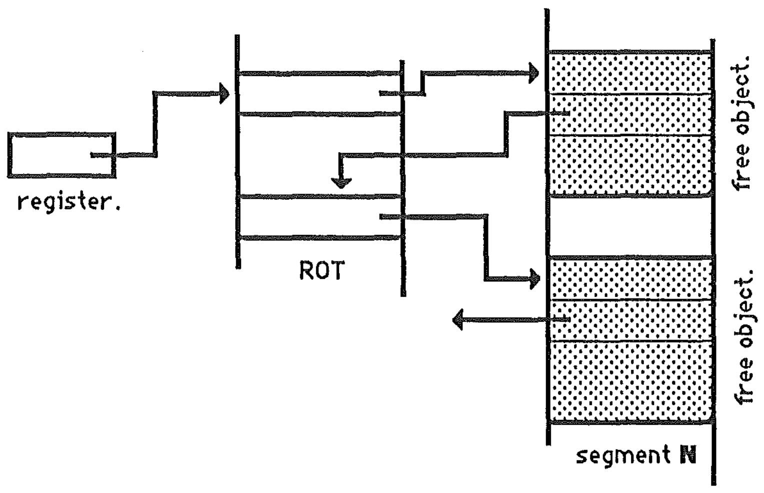

Figure 2.5: The arrangement of the free lists, connected via the objects class fields.

Unused object *pointers* are linked in a similar manner, one unused pointer, pointing to the next in the list. The end of this chain again contains a small integer value that cannot be an index into the ROT.

2.6 Allocation / Deallocation.

When space for an object is allocated, then the storage that the object represents is located at some point in memory. The actual location of the object is not known outside of the memory management unit, and all references to data in that object are via its object pointer. This allows the object to be relocatable in memory and thus functions such as memory compaction can be performed to help cope with fragmentation of memory.

Each object that refers to another object contains a copy of that object's object pointer. For each instance of a pointer to an object the object pointer's count bits are incremented by one. Thus if an object's count bits are greater than zero then the object is still referred to by some other object. When the count bits drop to a value of zero the object can be deallocated as it no longer has references to it.

The allocation of an object involves a search of the free lists of each segment for a free object of the correct size. If no object of the correct size can be found then the search is repeated, now trying to locate an object that can be split to provide an object of the size required and an object of some other size. Each of the segments in the address space is searched in turn. If no object can be found an error occurs and processing is halted by some means, or if it is provided, compaction of memory occurs.

Deallocation of an object simply involves placing the object freed at the head of the appropriate list in the correct segment. However, if the object contains pointers to other objects, we must reduce by one the count fields of those objects that are referenced by the deallocated object. If the count field of any such object becomes zero it must also be deallocated. This process continues recursively until all pointer chains have ended in non-pointer objects. Note that it is possible to have a circular list, and that this mechanism will not free space occupied in this manner.

2.7 The Interface.

The Smalltalk-80 system is, as stated above, divided into two functional parts. The first of these, the Memory Manager, is described above. The second is the bytecode interpreter. Communication between these two sections is via a set of interface procedures. These procedures are in theory the only method of communication between the two parts. Generally they have access to information that is hidden from the interpreter and perform quite simple functions involving specific combinations of the lower level routines of the memory manager. For example, the function **allocate** is a low level memory manager routine for the allocation of instances of objects in memory. The interface functions **instantiateClass:withPointers** and **instantiateClass:withBytes** both call the procedure **allocate**, but they do so giving different meanings to the length of object (words / bytes) and the values for the pointer and odd bits. Similar procedures exist for the access to objects themselves, as well as other procedures that perform differing interface tasks.

Chapter III

Early Design Decisions

3.0 Implementation Environment.

The machine on which we were to implement Smalltalk_80 was an Apple Computer Lisa 2/10. This machine is based on the Motorola MC68000 chip and comes with 1 megabyte of semiconductor RAM , a 10 megabyte internal hard disk drive and a 3.5 inch floppy disk drive.

The operating system is a UCSD P-system look-a-like called the Lisa Pascal Workshop. This system comes with a Pascal compiler and a 68000 assembler, a loader to combine various sections of a program under development as well as system utilities such as editors.

3.1 Object Pointers.

Object Pointers as described in section 2.2 are the even numbers 0,2,4 etc. of 16-bit unsigned integers, where the least significant bit is taken to be the type bit. The odd numbers 1,3,5 etc. represent small integers. As Smalltalk-80 was to be implemented in Pascal, where the 16-bit data type is the signed integer, it was decided to assign as the type bit, the sign bit of the Pascal integer type, bit 15. The result of this was that the positive integers including zero became the pointers to objects and all negative numbers became small integers.

The major reason for redefining the type bit in this manner was to avoid any problems that could occur with having both types of object pointer span both positive and negative numbers. With the numbers assigned in this new manner it meant that a Pascal FOR loop could be used in place of a more complex WHILE loop in most cases. This was an advantage as FOR loops were expected to be faster than WHILE loops when executing. However tests showed that a WHILE loop equivalent of a FOR loop was slightly faster.

Also testing to determine if a pointer was an object or a small integer was greatly simplified, small integers being all negative numbers.

3.2 Virtual Image.

The Virtual Image is a collection of objects and their associated ROT entries that have been derived from a functioning Smalltalk-80 system. Currently, Smalltalk-80 systems are written in the Smalltalk-80 language. The data that is contained in the Virtual Image is

- the compiled Smalltalk-80 code of the Smalltalk-80 system (it can be de-compiled to obtain the Smalltalk-80 code).
- all global and local variables of the system.

The Virtual Image was transported on tape from the Xerox Corporations's Palo Alto Research Center. This Virtual Image on this tape conforms to the standards set down in Goldberg and Robson for the Smalltalk-80 system.

3.3 Conversion of the Virtual Image.

Because of the changes made to the format of the object pointers as stated in section 3.1, the Virtual Image from the Xerox tape had to be converted to the format that was required by this implementation of the underlieing Smalltalk-80 system.

The conversion involved all 16-bit quantities in the Virtual Image that represented object pointers. Both pointers and small integers had to be converted from the form that they were in, to the form that using the new position of the type bit required them to be in. This meant that a new version of the entire Virtual Image had to be created. Words that had to be converted were:

- All objects that had their pointer bit set in the ROT that was to be found at the end of the Virtual Image on tape.

- All objects whose class was METHOD (class 11 decimal after conversion, 22 before). These objects contained a mixture of data to convert and data that was to remain unchanged (eg. bytecode).

Further details are covered in appendix 6 where a section of the conversion program is included.

3.4 The Resident Object Table.

As a result of the complexity of the Resident Object Table format detailed in section 2.4, it can be seen that to access some part of it such as the Free Bits, a series of manipulations involving AND and shift operations have to be used when setting, clearing or testing rather than some other simpler method of extraction. Further the segment bits are not conveniently accessible by simple machine level instructions.

To overcome what could be potential performance problems it was decided that

- each of the status bits would be separated into their own packed arrays of boolean with 8 bits per byte. Thus flags could be accessed by statments of the form IF Fbit[object] THEN...
- Because the MC68000 chip in use is byte addressed both the segment and count bits would be separated in their own arrays of byte sized objects to simplify access.

Also, as well as defining a separate array for the count bits, the range of the count bits would be extended from 0..127 with overflow at a count of 128 to a range of 0..254 with overflow occurring when a count of 255 was reached, ie. when all 8 bits were set. This was done for the following reasons;

- the test for 255 is no more complex than the test for 128
- it was hoped that by extending the point from where the count bits cannot be decremented (count bits "stick" at the upper limit and cannot be lowered) the amount of unrecoverable space that was lost due to the sticking of the count would be reduced.

It was also hoped that this increase in the count range would also to a large extent eliminate the need for a marking garbage collector. Wirfs-Brock of Tektronix [Wirfs-Brock 1982a] takes a somewhat different view in suggesting that the count bit field could be reduced to 3 or 4 bits as most counts fall within the range of 1..8. He does however state that this is dependent on whether units of this size can be accessed and manipulated efficiently by the machine being used, which is not the case with this implementation.

As can be seen in Table 3.1 taken after the initial 100,000 bytecodes had executed, there are a large number of objects (1,459) that require more than the 3 to 4 bits suggested, representing 7.9% of all objects allocated and 5.2% of the total space. With a larger number of interpreter iterations these figures increase to 19.2% of the objects allocated and 14.6 % of the space allocated. While the increase of the count does not lead to the reclamation of a lot of space, it is in no manner harmful. If we were to use the small count bit field, a marking collection scheme would have to be implemented. Thus we would have existing side by side two garbage collection collection methods, both serving to slow the system, where it is preferable that only one system is implemented.

limits of count bits (group)	Number in Group 100,000	Space in Group 100,000	Number in Group 1,000,000	Space in Group 1,000,000
1	13,056	218,143	11,690	202,284
2..7	3,922	42,027	3,202	33,998
8..15	638	6,699	1,767	21,264
16..127	631	6,017	1,471	15,595
128..254	46	399	105	1,139
255	144	1,107	200	2,357

Table 3.1: The number of objects falling within certain ranges of Count bits, and the space occupied by them after 100,000 and 1,000,000 bytecodes executed.

Thus the final format of the ROT decided upon was:

- 3 packed arrays of boolean for the F, O and P bits.
- 2 arrays of bytes for the segment and count bits.
- 1 array of words that holds the locations of an object in a segment in the same manner as described in section 2.4.

3.5 Memory Limitations.

The Pascal compiler, linker and memory management system of the Lisa divides memory into 32k byte segments. There are two consequences of this:

- The total global memory available to any Pascal program is limited to 32k bytes. As a result, the only parts of the Smalltalk-80 ROT that could be placed into this area of memory was the 3 packed arrays of boolean used for the flag bits (the ROT in Goldberg and Robson uses one complete segment = 128k bytes).
- The Pascal compiler will not allow any data structure of a size greater than 32k bytes to be declared. Thus a memory segment which is 128k bytes in extent had to be split into four parts. Each part being allocated on the heap via the execution of a Pascal NEW statment (heap also suffered the 32k byte limitation).

The Lisa computer in its initial configuration had a total of one megabytes of random access semi-conductor memory. Of this, just over one quarter (300k bytes) is available for the stack and heap space. Half of the total RAM memory in the system appears to be reserved for the exclusive use of the operating system.

This allowed the allocation of only two memory segments in RAM. However the Smalltalk-80 system requires a minimum of 512k bytes in its address space because of the extent of the Virtual Image. This meant that it was necessary to allocate the remainder of the memory space required on the hard disk and use a virtual memory scheme. Thus the memory space of the system comprised two segments allocated in RAM and four or more segments allocated on the Lisa's internal hard disk. The organisation of virtual memory is detailed in chapter 4.

3.6 Recursion.

In the previous section the small amount of heap and stack space remaining after the system data areas had been allocated was noted. Because of the uncertainty that existed about the amount of memory that would be used by the stack, all recursion that existed in the Smalltalk-80 memory management code was removed. This then required that another, non-recursive method of coding the procedure **countDown** be used, this was done and is detailed in chapter 6.

3.7 Other Changes.

The only other section of code to be written in a different manner from the book was the code that allocated space for objects to be allocated. To keep the code as simple as possible, so as to aid a quick implementation, initially only one list of free objects was used per segment. In addition the order in which segments were search was changed so that RAM memory was always given priority over disk when a new object was being allocated.

Chapter IV

Implementation of Virtual Memory

4.0 Introduction.

This chapter describes the Smalltalk-80 system as it was implemented on the Lisa with a total of only 1 megabyte of RAM. Because of the small number of segments that were able to be allocated with this amount of RAM, a virtual memory approach was adopted so that the complete Virtual Image could be contained in the address space of the system.

4.1 The Need for Virtual Memory.

The initial Smalltalk-80 system was comprised of a total of 1 megabyte of RAM. Of this, half appeared to be permanently allocated for the use of the Pascal Workshop system. This conclusion was reached after repeated attempts were made to allocate more than 300k bytes on the heap of the system. The behaviour of the Workshop system tended to support this. For example, when re-entering the editor there was no noticable disk activity or time lag; this seems to show that the editor remained in memory while "user" programs were running . Also files that were left in the editor's workspace remained there, there was no time delay when reediting these files. However the editor can be explicitly removed, unlike the rest of the operating system, and appeared to have very little effect on the amount of usable heap. Of the remaining space not all could be used to allocate memory space for the Smalltalk-80 system. The reason for this is that a certain amount of space is required for the system tables and the code of the system's lower levels. Further, as the Smalltalk-80 system memory is being allocated in the heap / stack area, space must be left for the growth of the stack during execution. Thus virtual memory was examined as a solution to the space shortage problem.

Virtual memory management in Smalltalk-80 requires the study of two related problems. The first is that objects must be brought into memory from disk and allocated space in RAM just as if they were new objects. Here it is necessary to accertain that the object does not exist in the RAM portion of the Smalltalk-80 address space. Once this is known, it is essential that the object can be accessed quickly, space allocated for it and have it brought into RAM.

The second is that objects must also be returned to disk when the amount of RAM available for the allocation of objects falls below some limiting amount. Here, the principal concern is that of beingable to decide which objects to move out to disk, and of being able to do so quickly.

4.2 Addressing Considerations.

In computer systems where virtual memory is implemented, such as the VAX (Virtual Address eXtension) series of machines, when a unit of virtual memory storage, in this case a page of 512 bytes; is transferred from disk to RAM a copy is kept on the disk. If a page in RAM remains unaltered during processing then, when the system decides to remove it from RAM so that another page may be brought in, the page does not need to be copied back to disk [Hwang 1984].The fact that a page in RAM has been altered is recorded by a "changed" or "modified" bit. As most of the work involved in implementing virtual memory in large systems is done using hardware, good performance in these systems has been achieved. As it is not possible to implement virtual memory in hardware with the Lisa system, it must be implemented in software. Because of this it was necessary to produce a system that did this with the smallest amount of overhead possible.

In this Smalltalk-80 system virtual memory was treated in the following manner:

- it was decided that all processing of objects should occur with the object resident in a RAM segment
- copies of objects in RAM would not be kept on the disk segments. The reasons for this are detailed below

- The memory organisation for the disk segments would be exactly the same as for the RAM segments so the same code could be used for both. Thus moving an object from disk to RAM creates a free object in a disk system

Disk memory was handled in this way because of the overheads in keeping objects both in memory and on disk at the same time. These overheads break down into two classes:

- extra code would be required if a changed/modified flag was to be implemented (a "used" bit was introduced after these decisions were made)
- there are problems connected with addressing if there are two copies of an object in existence at the same time

Considering the first point when a RAM copy of an object is changed so that it differs from the copy that is on disk then we must remain aware that this has happened. This means that on every call to the procedures **storePointer**, **storeWord** and **storeByte** we must set a flag to show that the object has been modified. The actual amount of code required is small. What is of concern is the number of times that the code would be executed. In the first 100,000 bytecodes there are 162,444 calls to **storePointer** which uses 11% of the time required for a run of this length.

On the second point, when we keep two copies of an object we have to find some method of associating the object that is in RAM, which will always be the most recent version of the object , with the copy of the object in a disk segment. As stated before, all accesses to an objects are via its pointer. Here we have the situation of either:

- having two objects referred to by the same pointer value (doubled addressing). What we have here are two sets of address for each object. The situation of being in two places at the same time. One set of addresses would be be the RAM address of the object, the second would be the disk address of the object. Thus if the RAM address was non-valid then a fault would occur.
- having two objects that have different pointer values but which are in fact the same object at two different periods in time (doubled pointers).The problem here is to map one object pointer onto another. This could be done be done by

allocating in memory, one extra word for each object pointer, this extra word would hold the object pointer of the copy on disk. So that for every object pointer of an object in RAM we have a copy of the object pointer for the disk object.

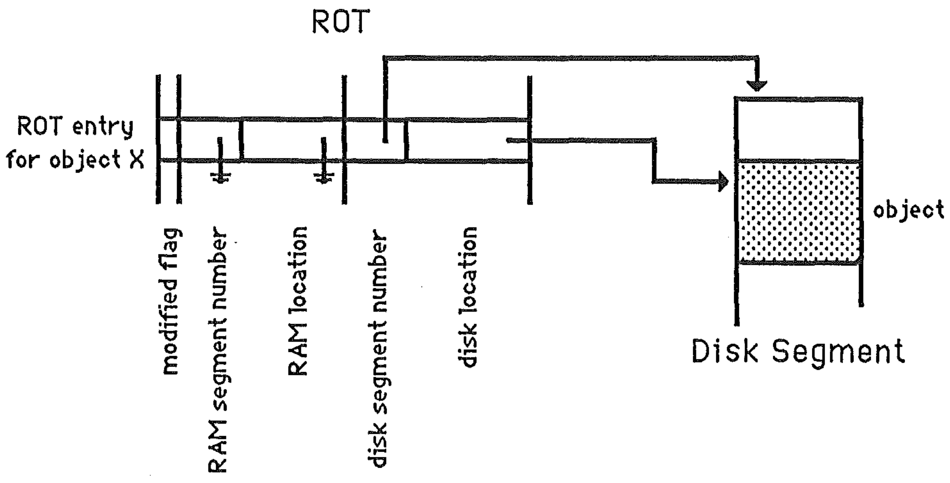

Figure 4.1: The layout of the ROT for the double address method before the object X is copied into RAM. Note that the two parts of the RAM address have nil or invalid values.

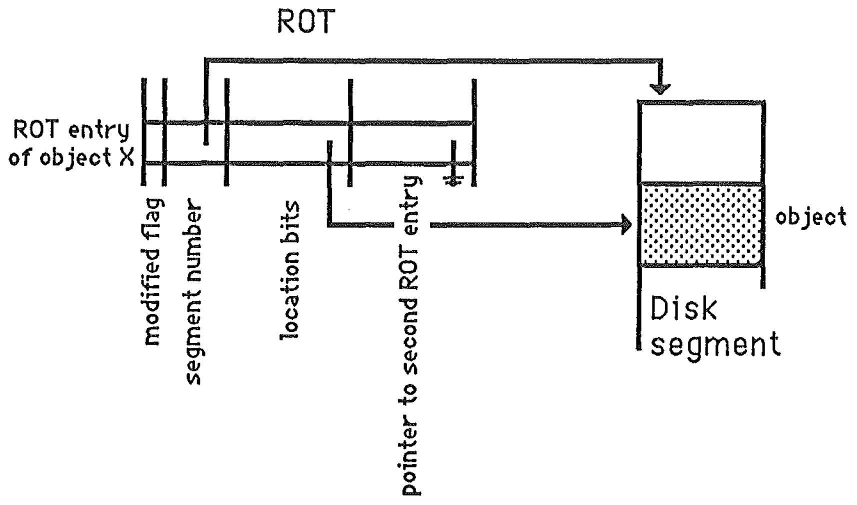

Figure 4.2: The layout for the doubled pointers method before the object X has been moved to RAM. Note that the pointer to the RAM address is nil.

To clarify what is being proposed here, consider the example of an object X residing in a disk segment. The state shown in figure 4.1 is for the case of double addressing and figure 4.2 is the case of doubled pointers. At some point the interpreter makes a reference to this object. When this access is made an "object fault" (from page fault) occurs. This fault causes the object to be brought into memory.

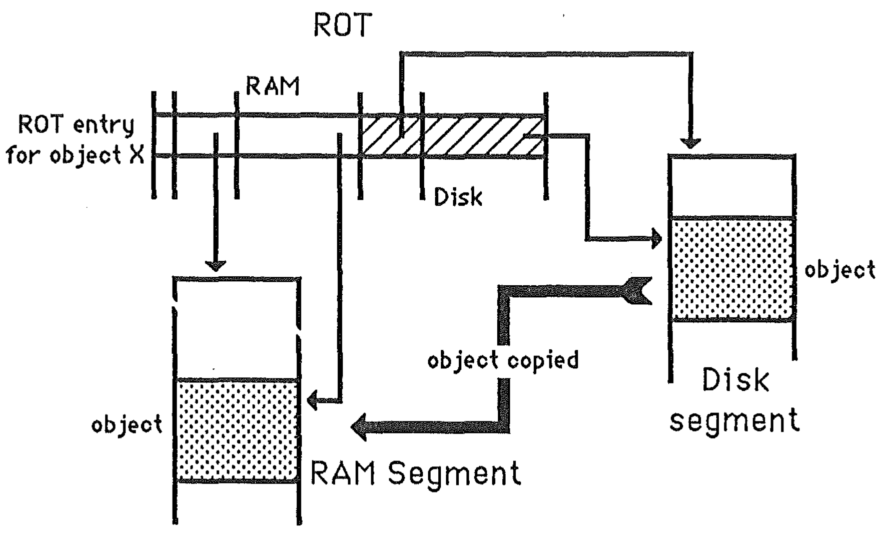

Figure 4.3: The form of the ROT entry for double addressing after the object **X** has been copied into RAM. As there is a valid RAM address, it will be used to access the object.

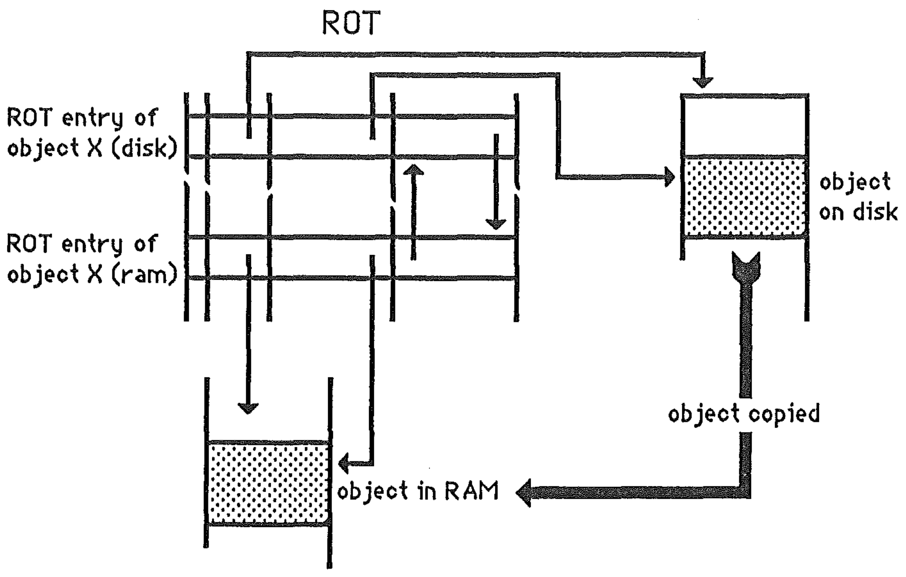

Figure 4.4: The form of the ROT for the doubled pointers method after object **X** has been copied from disk. It should be noted that as all objects start in RAM (section 3.5) access will be via the RAM ROT entry.

Figure 4.3 shows the state of the ROT for double addressing after the object X has been moved into RAM. At this point the second set of segment and location fields get non nil values that give the location of the object in a RAM segment. If another object that did not have a copy of itself on disk existed, then it would have nil values in the disk segment and location fields. That an object was not in RAM would be shown when it was found that the RAM segment field had a nil value. Strictly speaking the location field cannot have a nil value as all 16 bits are used.

Figure 4.4 shows the state of the ROT in the doubled pointers case after the object fault has occurred. Here a second ROT entry has been allocated to hold the address of the object in RAM. The extra field in the ROT is used to form a link between the two objects. Access to the object in RAM must now be made via indirection from the original pointer (shown here as the disk pointer) to the new pointer. If the object had existed in RAM before being moved to disk then the indirection would occur in the opposit direction.

There are two main objections to using these methods. The first of these is that they are complicated. To implement either of the methods above requires extra code, which would slow the system. Also, the levels of faulting (figure 4.9) when compared with allocation (figure 4.7) and deallocation (figure 4.8)*seemed* to indicate that the faulting rate would have little effect on the system.

The second and major objection is that there is insufficient space available in memory for the extra data structures that would be necessary to hold the data required to implement these techniques. It was noted in section 3.5 that there was very little space remaining after the allocation of system tables and RAM segments. Only 58k words remained in the system heap/stack space for allocation. Of this:

- 2K bytes would be required for the changed/modified flags if they were allocated as a packed array of boolean.
- If the segment and allocation bits were to be allocated for a second time, a total of 48K words would be required, 32K for the location bits and 16K for the second set of segment bits.
- If the second method using double object pointers were used only 32K words would be required for the array of pointers to their ROT entries.

As the heap shares space with the stack, there is some doubt as to whether enough space could be made available for the extra data structures without having the stack grow into them. The Lisa's operating system itself recognises the problem and will not allocate heap/stack below a certain limit. With 68K words of stack/heap remaining on the system with 1.5 megabytes of RAM, another 64K word segment could not be allocated. As the total amount of space remaining is similar in this case to what would have been left if the extra 48K words needed for double ROT entries had been required, it seems probable that operating system would not have allowed the allocation to occur.

4.3 Format of Disk Segments.

It was decided to treat disk memory in exactly the same manner as RAM segments. That is, it would be randomly accessible and it would have the same logical organisation as the RAM segments. This would then allow for access of objects in the same manner as in RAM and allow the use of some of the same code. A similar configuration exists in the LOOM virtual memory system [Kaehler 1982].

Disk segments were implemented by using random access files of the type integer, access to each file being made by the **seek** procedure built into the Pascal Workshop System. The **seek** procedure takes as its argument a 16 bit signed integer which is the physical number of the record being accessed. As an integer is used, this limits the number of records in any one file to 32768 elements. As there are 65336 integer (word) elements in a Smalltalk-80 segment, at least two files would be required to hold one segment.

At an early stage of this work, when disk memory was being implemented, RAM memory was being accessed as a series of four 16k word (32k byte) blocks. For this purpose there were a set of procedures that translated 16 bit integers into a form suitable for access in this manner. Because of this, each disk segment was split into 4 sections. The high order 2 bits of the location field identified which file each quarter of a segment was in and the low order 14 bits giving the offset within each file.

Each set of four files that made up one segment was distinguished by its segment number. An array of file descriptors with an index range that matched the segment numbers assigned to the disk segments, was accessed by the functions **HCObyte** etc., during access to the disk segments.

4.4 Buffering.

It was decided that no buffer would be used for disk transfers, that is all access to objects in disk segments would be done in a completely random, word by word manner. There are two major reasons for doing this. The first is, that unlike languages such as Pascal or C, there appears intuitively to be very little locality of execution in the Smalltalk-80 system.

This assumed absence of locality is due to the manner in which objects are allocated. Unlike commonly used programming languages, there is very little "static" structure to a Smalltalk-80 program. In addition the "stack" in Smalltalk-80 is allocated as objects, one for each active method. These are the active contexts, and because of the rate at which they are created and destroyed they can spread the site of execution over a large area (active contexts are objects of size 20 and 40 words, see figures 4.7 and 4.8).

This effect has been noted by Ballard and Shirron [1982] where they state:

> "We were interested in determining the performance of paging for a system like the Smalltalk-80 system, where the locality of reference between adjacent objects is much smaller than in traditional language environments."

They go on to give some details of the effects of their Smalltalk-80 memory management system running under VMS on a VAX 11/780. They found they were getting about 250 page faults per second (512 byte pages) during browsing or compiling with a working set size of 512 pages. However VMS allocates some core as a page buffer which reduces the number of times the disk is accessed. So with an unloaded system they could effectively use all of main memory. They also note "the excessive paging we encountered when we have less than about 3Mbytes of real core for our use."

The other major reason why a buffer is not implemented is the small average size of objects in the Smalltalk-80 system. For the Virtual Image, the average size of objects is just over 14 words. This, with the extremely skew distribution of object sizes in the Virtual Image (figure 4.5) and the distribution of objects allocated in the first few hundreds of thousands of bytecodes (figure 4.7), shows that on average small objects are the frequently most used and thus faulted (figure 4.9). Considering this small object size, and the probablity that any buffer used would have to be refilled with new information at many points, shows that buffering would have been severely detrimental to performance.

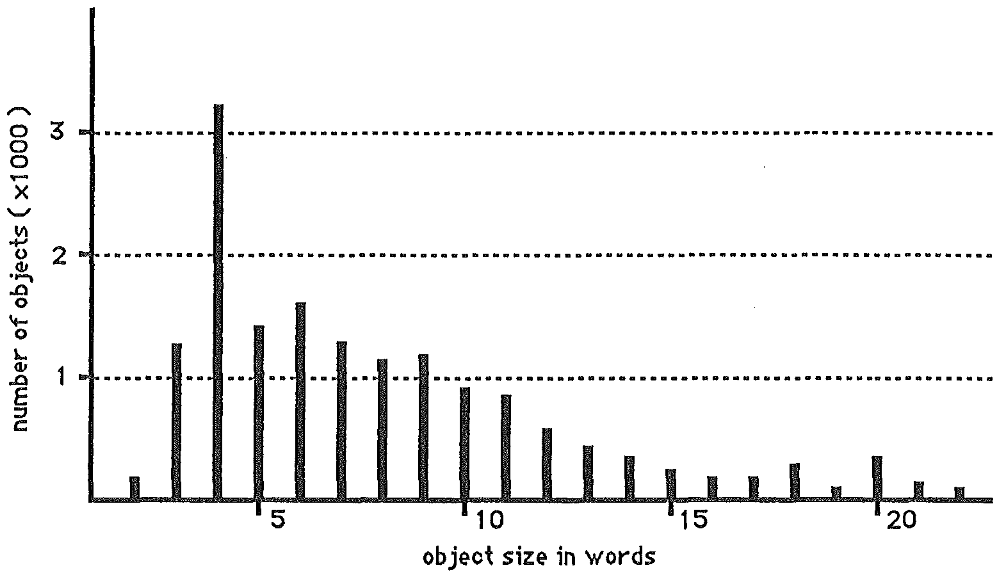

Figure 4.5: The distribution of initial object sizes in the Smalltalk-80 Virtual Image.

Further Table 4.1 below shows the time taken to read 64K words from a file of type integer using different methods of access. The first of these is on a word by word basis using the **seek** procedure. The remaining results were obtained by using the Pascal **blockmove** procedure to transfer a specific number of 256 byte blocks from disk to memory.

Transfer size	Number of blocks	Time Taken in mSeconds for 64k words.	Expected Time to Transfer 20 words. (extrapolation)
1 word	N/A	19380	6 mS
256 words	1	14131	55 mS
512 words	2	7394	57 mS
1024 words	4	5956	93 mS
2048 words	8	4791	149 mS

Table 4.1: Transfer rates from disk to memory, timed data for transfer of 64k words, with the transfer time for 20 words calculated.

Unless there is a large locality of objects in the virtual memory segments of the Virtual Image, only very small gains could be expected in any buffering of disk traffic. More probably no gains will be made because of the added complexity introduced.

4.5 Image Load

The first phase of a run with the Smalltalk-80 system, after the initial setting up of memory and register values, is the load of the Virtual Image into the system. The data for the load is contained in two files. The first of these consists of the initial objects resident in the system in the converted format described in section 3.3. The second file is a version of the ROT that comes with the Smalltalk-80 distribution tape [Xerox 1983a].

The ROT that came with the Xerox Smalltalk-80 distribution was read by the program that performed the conversion of the Image into the form that was to be used. The ROT itself was converted from a series of 32 bit records of the form given in Goldberg and Robson to a series of Pascal records. This enabled the the manipulation of the count and bit fields to be made in advance of the actual Smalltalk-80 image load by a Virtual Image conversion program. The aim of this exercise was to reduce the time taken to complete a load.

The method used during the load is to read the ROT entry for the next object pointer from the file containing the ROT. This information was then used to allocate the object space in Virtual Memory. The words of data that make up the body of the object can then be read from the Virtual Image file and transferred to the space allocated. The object is then resident in the Virtual Memory space of the Smalltalk-80 system.

It should be noted here that while the ROT is complete, in that there is one ROT entry for each possible object, the Virtual Image supplied by Xerox does not have a complete set of objects, both free and allocated. All the objects that are marked as being free in the ROT are not included in the distribution copy of the Virtual Image.

The first loads of the system were made by treating disk memory in exactly the same manner as RAM. Here the Smalltalk-80 system would start by allocating objects in RAM. When this space was exhausted, allocation would then continue in the disk portion of Virtual Memory. A load using this manner of allocation took approximately 50 minutes.

It was found that the bottleneck was the moving of large amounts of data to the disk segments in an object by object manner, that is allocating an object space in a disk segment, then reading the data for that object from disk and then moving the data word by word via the Smalltalk-80 function **HCOwordPut** to the object.

This problem was avoided by moving whole RAM segments to disk as a unit of 64k words. This was done by using the Pascal Workshop procedure **blockmove**. Here the objects are still allocated in the same manner as before, with RAM and some disk segments being used in the same manner. The change was that when segment 0 became full it was moved as a unit to a disk segment that was unused. This move operation taking around 32 seconds to complete. Allocation would then resume with the majority of new allocations occurring in segment 0. The effect of this change was to reduce the time taken for a load to 7.5 minutes.

4.6 The Basic System.

The condition that gives rise to an object fault occurring is the detection that the object resides in one of the disk segments allocated for virtual memory. The test for this condition occurs at every access to an object. If the condition is met, then space is allocated in RAM and the data associated with the object is copied from disk. The copy on disk then becomes a free object and and the access is completed. The state of memory after the movement is shown in figure 4.6 below, the pre-fault state being similar to Figures 4.1 and 4.2.

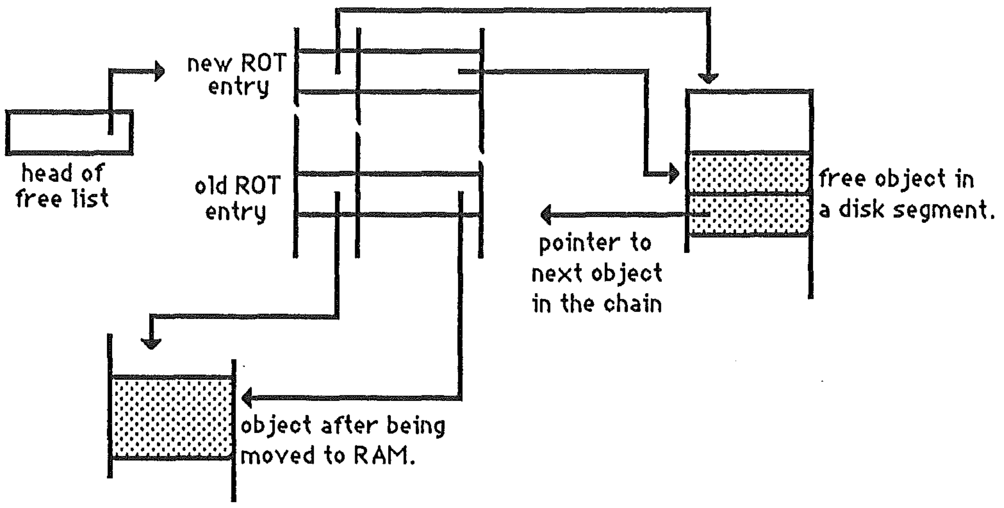
new ROT entry
head of free list
old ROT entry
free object in a disk segment.
pointer to next object in the chain
object after being moved to RAM.

Figure 4.6: The state of the ROT table and the RAM and disk segments after object X has been faulted from disk to RAM in the implemented system.

A transfer in the opposite direction occurs when the RAM allocation routine fails to find sufficient space to allocate an object. When this occurs, space is freed in RAM by moving the object(s) to disk in accordance with some suitable algorithm.

4.7 System Aims.

The major aims of the system are to:

-be able supply the interpreter with any memory that it may request

- provide access to objects that are allocated for the interpreter and
- dispose of objects that are no longer in use. This is done so that the first aim may be met.

As a secondary goal to the first point noted above, it is necessary to locate objects to be moved from to disk because of the limited amount of RAM available. Several differing methods were examined to provide this facility in a manner that was:

- fast enought to be transparent to any user of the system and

- did not introduce a level of object fault occurrence that also resulted in reduced performance of the system.

To achieve these aims two strategies were examined. The first of these was to try and lower the number of faults that required the movement of objects. The second was to examine a number of different methods for locating and moving objects to disk segments.

4.8 Fault Reduction.

The aim here was to reduce the number of objects that are moved from disk to RAM. As stated in section 4.6, a fault occurs whenever an object is accessed that is resident in a disk segment. What was attempted here was to delay the movement of objects from disk to RAM.

As an alternative to moving an object on its first access, the move was delayed until a second access occurred in succession with the first. This was done by allocating a global variable to keep track of the pointer of the object that had most recently been accessed.

The application of this method gave results that were indifferent that is, there was no significant change in the the time taken to complete a run, or in the number of faults that occurred. Because of this, and because of the added complexity of the code required to implement this method, it was discarded as a viable means of improving performance.

4.9 Allocation Strategies.

The method described in Goldberg and Robson for locating space to allocate an object is:

```
FOR each segment
    search the segment's free list(s) for a suitable free object.
    UNTIL allocation occurs OR every segment has been seen.
```

This is the basic structure that was retained for the virtual memory implementation. One small change was made to this system. The change was that the segments that were searched when allocating an object were restricted to the RAM segments, unless an object was being removed to disk.

If all segments are searched and allocation has not taken place, then the segments are examined again sequentially after a compaction step has been performed. No action is specified by Goldberg and Robson if this further step again fails to locate space for allocation of an object.

The order in which segments are searched is determined by what segment a successful allocation last took place in.

If allocation **n-1** took place in segment **N**, then segment **N** will be the first segment searched when allocation **n** takes place. If allocation **n** is unsuccessful in segment N, then segment **N+1** will be examined. If segment **N+1** does not exist, then segment **0** becomes the next segment.

The decision to allocate new objects exclusively in RAM was made by considering that an object was being brought into existence because some amount of processing was to take place on that object immediately. If this is the case in general then, if an object was initially allocated in a disk segment, any further processing would trigger an object fault. Thus RAM allocation would occur some small time in the future, resulting in two related allocations occurring where one would have sufficed.

4.10 Removal Strategies: Simple Searching.

As was noted in section 4.9, the allocation scheme given in Goldberg and Robson provides no course of action in the event that an object cannot be allocated after the compaction phase has been executed. To implement swapping to disk, a further stage was added to create sufficient space within RAM to allow the allocation of an object. If more than one object has to be removed from memory to provide sufficient space for the allocation, then a compaction pass is made prior to allocation.

The methods examined initially involved the search of a segment (via the ROT) and removal of selected objects. The first algorithm used, 4.1, was a sequential search of the RAM segments to find an object that was (in the order tested):

- in the segment where we were attempting the allocation,
- allocated. This was checked to avoid re-examining objects already seen via the free lists. Also there is no point moving a free object to disk! and
- of sufficient size to satisfy the request for memory. This test was made last because access to an object's size field involves indirection to the object itself; the other information can be obtained from the ROT.

The method given above takes no account of how often an object is being accessed. One of the largest objects in the system is the method dictionary, which contains over 4000 words. The removal of this object would cause an object fault almost immediately. Further, it was expected that some of the largest objects in the system would be the screen objects. These are areas of system memory used as bit-maps for the graphical output device.

This would introduce the problem of a large and/or frequently used object being moved out of RAM only to be returned quickly, presenting a larger allocation problem. To remove this problem another flag was introduced to mark objects that had been used in the recent past. This U flag was set on every access to an object and periodically reset. The period used for the reset being N ticks, where a tick is defined as having occurred at the start of each allocation. The concept of a tick is based on work done by Oldehoft and Allen [1985] on adaptive exact-fit storage management. As runs using the system are identical for the first 60,000 bytecodes (all runs use the converted Xerox Virtual Image), every object that is allocated in one run will be allocated at the same point in the execution of the system in the next. If the events occur closer together (in time) in run x than in run y, then we can say that run x was faster than run y. Because of this, the average value for ticks can be used as a measure of system speed.

The code of the second algorithm 4.2, a straight development from 4.1, was:

```
function find_to_swap( size,              { min. size of object to be removed }
                       first,             { where to start the ROT search }
                       last : integer     { where to end the ROT search }
                     ): integer;
label 666;
var
    i : integer;
begin
    find_to_swap := Null;                         { a non pointer value }

    for i := first to last do
       if segbits(i) = currentSegment then        { it is in the current segment }
          if not Fbit(i) then                     { and it is allocated }
             if not Ubit(i) then                  { and it has not been used recently }
                if sizeBitsOf(i) >= size then     { and it is big enough }
                begin
                   find_to_swap := i;             { return the identity of the object }
                   goto 666;                      { this is faster than redoing the FOR}
                end;
666:
end;
```

Both the methods detailed above are contained in an outer loop. This outer loop keeps track of how much space has been made available in the segment, and continues until enough has been freed for the allocation. Also, on each successive iteration it reduces the size of object required by a factor of 2 if no object of the size requested is removed. This was done to ensure that an infinite loop would not be entered into.

The third algorithm, 4.3, is described as:

- to use algorithm 4.2 to try and locate *one* object that can be moved to disk. If this fails, to then move through the ROT removing *any* objects that can be removed until the required amount of space has been freed.

The procedure used to do this was **move_emmass** shown below.

```
procedure move_enmass(we_wants : integer        { total amount of space required }
                             );                 { var in honour of the Gollum }
var
    i : integer;
begin
    i := 128;  { no use looking at the first few objects as they seem to be used often }

    while (seg_free[currentSegment] < we_wants) { while space is still needed and... }
          and (i < K32) do                      { we are not at the end of the ROT }
    begin
      if segbits(i) = currentSegment then
        if not Fbit(i) then                     { it is allocated }
          if not Ubit(i) then                   { it is not in use at the moment }
          do_a_swap(i);           { does actual swap of object i from RAM to disk }
      i := i + 1;
    end;
end;
```

4.11 Search Performance.

The methods used to locate objects to move out of RAM to disk described above were implemented in the order in which they were given. Neither of the first two techniques was found to be effective in freeing space for allocation when used alone (see table 4.2).

However the last method given (4.3), showed a marked improvement in the time taken to allocate large objects. As an example, consider the manner in which the interpreter functions. The interpreter cycles in a loop; it gets the next bytecode to be executed and then performs the function required. After 720 bytecode fetch cycles the interpreter asks the memory manager to allocate an object of 19,200 words in extent (this is the screen display). The best performance that could be obtained using the first two methods alone was 48 minutes to clear sufficient space for the object to be allocated.Using the last procedure given, **move_enmass**, this time was lowered to around 8 minutes, an improvement of a factor of five.

4.12 Anticipatory Removal.

The need to distribute the overhead of removing objects from memory (RAM) can be seen from the large length of time that it takes the systems described in section 4.10 to allocate a large object.

Much work has been done in the area of paging systems, and their behaviour is quite well understood. In an "optimal" system the reference string generated by any run will be known in advance. As it is known what pages will be accessed, it is possible to remove pages that will not be required to make room for those that are [Denning 1970] in advance. Most systems approximate this by removing pages according to some scheme such as "least recently used" when a page fault occurs. The Smalltalk-80 system as described so far performs in a similar manner: objects are only removed when space is required. In an attempt to distribute the removal of objects, it was decided to examine systems where space was continually being freed during execution, hopefully coming closer to the Denning ideal and in effect attempting to predict what will not be needed.

The basic manner in which this was done was as follows.

- Every time that a new object is allocated, an object that is already in RAM is moved out to disk, whether RAM is full or not.

Several versions of this basic idea (algorithm 4.4) were implemented. The major variants include the following.

4.4: Move an object out of RAM every time that a request to allocate space is received.

4.5: Move an object out of RAM every time that the interpreter requests that a new object is to be allocated. This is opposed to version 1 where an object fault would trigger the swapping out of an object.

4.6: Use the method described in 4.5 with the following modification: in both 4.4 and 4.5 the search for objects to remove begins at the start of the ROT and continues until either an object is found that can be removed or the highest *used* ROT entry is reached. Here a variable was introduced to keep track of the point reached in the previous search. The next search of the ROT then takes place from beyond this point.

4.7: Versions 4.4 through 4.6 all take into account the segment where the object resides, using code similar to that given for procedure **find_to_swap** in section 4.10, with the test for size removed. This algorithm disposes with the check to see if the object being removed is in the current segment.

These techniques are used in addition to the methods described in section 4.10. The methods detailed there are only used again when it is found that an object cannot be allocated.

4.13 Performance of Continuous Removal.

Table 4.1 below shows the relative performance of six of the methods described in sections 4.10 and 4.12 over the execution of 750 bytecodes (the main bottle-neck is at 720). The columns of the table represent:

- the average length of each tick in seconds,
- the maximum length of any tick,
- the average number of faults generated between any two ticks and
- the average number of nodes of the linked lists of free objects examined for each allocation.

Algorithm Used.	Average Length of Tick in seconds.	Max Length of Tick in seconds.	Fault Rate per Tick.	Average Number of Nodes Seen.
4.2	48.4	2923	1.8	30
4.3	19.4	955	1.6	49
4.4	19.4	575	2.2	73
4.5	13.7	587	2.0	54
4.6	12.0	538	1.9	46
4.7	11.4	543	1.8	76

Table 4.2: Statistics for some implementations of virtual memory.

The first entry in table 4.2 comes from a run using the second method listed in section 4.10. It can be seen that this method is notable for the maximum length of a tick, over 3 times greater than the second largest value. The average length of a tick is also correspondingly large.

With a move to a method that applies more "brute force", it can be seen from the second row (4.3) that the last technique given in section 4.10 provides a large improvement over the first. This can be explained by noting that the procedure **move_enmass** does fewer scans of the ROT when finding objects to remove.

The remainder of table 4.2 (entrys labeled 4.4 to 4.7) shows performance figures for all the methods given in section 4.12. It can be seen that a large initial improvement in the speed of execution has been obtained. This is due to having reduced the work that the procedure move **en_mass** is required to do when getting the total amount of free space to a level where the object can be allocated.

The further changes being made to the system, while resulting in continued improvement of execution speed, offer no large gains. In effect what we are now doing is tuning of the system implemented by method 4.4.

The third column giving the average number of faults per tick shows that this figure has remained relatively constant (on order of magnitude) over the changes being made to the system. The peak introduced when anticipatory removal is first introduced is removed later by changes involved in the tuning of the system.

4.14 Fault Rate.

Examination of Table 4.2 shows that, while the execution speed has increased, the number of nodes examined to allocate an object has also shown a corresponding rise. This has occurred becauseonly one list of free objects per segment is being used.

The effect can be understood by considering the following. The procedure that removes objects regardless of the amount of RAM free does not take into consideration the size of the objects that it removes. As these measurements are being made at an early stage of the system's execution, reference to Figure 4.5 shows what the distribution of object sizes in the system at this time can be expected to be. Figure 4.7 shows the measured distribution of objects allocated over a spans of 100,000 , 200,000 and 300,000 bytecodes. As can be seen from Figures 4.7 and 4.8, the vast majority of objects allocated are of a size of 20 words (note that the vertical scale is log base 10). The distribution of objects resident in the system has its peak at a much lower value.

Early removal of objects is creating a large number of smaller spaces at the start of each linked list, where they are added. The interpreter is requesting objects of a larger average size than those being freed. The result is that more nodes must be examined to locate objects of significant size.

Because of this, a second free list was added for each segment. The way the two lists were organised was that all free objects whose size was greater than some limit N were placed on one list. All other objects, whose size was equal to or less than the limiting value, were placed on the second list. Experiments were conducted to determine the "optimal" value for N, the cut-off point. This was determined from Table 4.3 below to be at the value of 13. As can be seen from Tables 4.2 and 4.3, the introduction of two lists resulted in the desired effect: the number of nodes examined was reduced and the Smalltalk-80 system made faster.

cut-off point N	seconds per Tick.	average number of nodes examined.
one list	6.3	82.6
5	6.1	55.9
10	6.0	35.3
12	6.0	28.0
13	5.9	25.7
14	6.0	25.6
15	6.0	24.4
20	6.0	22.2
30	6.1	58.9
50	6.1	62.7
100	6.1	60.4

Table 4.3: The effect of the cut-off point N on speed of execution.

Some of the results of having two lists are shown in Table 4.4 below; also shown is the effect of increasing and decreasing the rate of anticipatory removal on the system.

Rate of Preemptive Object Removal per Tick.	Seconds per Tick.
2.0	35.3
1.0	3.4
0.5	2.7
0.33	2.8
0.25	2.9
0.2	2.9

Table 4.4: The effects of the rate of preemptive removal.

It can be seen that the movement of objects out of the RAM space has resulted in an increase in speed of execution by a factor of four. Also it should be noted that a balance must be met between removing too many objects, as demonstrated when two objects were removed per tick, and too few were removed as in the last case.

4.15 Paging.

Despite the gains in performance introduced by techniques such as finding and moving objects en masse and by the use of anticipatory removal, the performance of the system was such that the system could be used for development, but was of no value for practical use.

A breakdown of the problem here shows that what was being attempted was the management of *physical* memory by using logical entities, ie objects. Hwang [1984] states that "segmentation is a technique for managing virtual space allocation, whereas paging is a concept used to manage the physical space allocation", pages being the segments in this case (see section 2.3).

Investigation found that most of the time consumed by the system was in :

- locating objects to move to disk so that free space was produced in RAM and
- compacting the RAM segments so that what free space was available could be used.

Because of this, the system was changed so that instead of having a system where virtual memory was implemented wholly with objects. A system was produced where, the object is still the unit for movement from disk to RAM, but the segment is used for movements in the opposite direction. As was noted in Section 4.2 when loading is being done, removal of *entire* segments from RAM to disk takes approximately 32 seconds. This code for moving whole segments (128k bytes) was the basis for paging like system, where we know how much space we need on disk. The page size here is one segment. The algorithm for allocation was modified to become:

```
if seg_free[1] < 100 then                 { seg_free holds TOTAL amount of free RAM }
    moveSegment(1);                       { move segment 1 to disk as nearly full. }

if size > 50 then                         { test for split between segments }
begin
    obj := listSearch(0, size);           { do the allocation in segment 0 if possible}
    if obj = Null then                    { if object null the allocation has not occurred }
    begin
        moveSegment(0);                   { move segment 0 to a disk segment }
        obj := listSearch(0, size);       { do allocation is segment 0 now it is clean }
    end
end
else                                      { object is smaller than 50 words }
begin
    obj := listSearch(1, size);           { search seg 1 to allocate the object }
    if obj = Null then
        listSearch(0, size);              { if cannot allocate in seg 1 allocate in seg 0 }
end;
```

What should be noted here is that objects had to be further divided into two classes based on their size. This was done because large objects such as the method dictionary (4,000+ words) and screen objects (19,000+ words) are expected to remain in the system for a longer period than objects of smaller size (contexts at 20 and 40 words). Also, as stated in section 4.10, if a large object is removed it requires a much larger effort to bring it back into RAM than does a small object. Therefore it is desirable to keep the movement of sizable objects out of RAM to an absolute minimum.

The decision to split the objects being allocated in each segment at a size of 50 words was made by examining the data shown in Table 4.5

value of split	time for a run of 250,000 bytecodes in seconds.	free space in segment 0 at end (RAM) in words.	free space in segment 1 at end (RAM) in words.	number of disk segments used.
1000	1563	45928	52766	8
500	1567	44569	55193	8
100	1473	36984	19487	7
50	1404	32193	35273	7

Table 4.5: The effects of different splitting levels. Values smaller than 50 were not examined as contexts have a size of 40 and are allocated / deallocated relatively frequently.

It can be seen that the last two limits examined for split use less disk space than the two methods using the larger cut-off limits. Also note that there is a small but steady decrease in the length of time taken for each run, with a total improvement of 159 seconds, this being three minutes or 10% of the run time. As the major emphasis is on producing a usable system, speed is considered the most desirable factor after correctness. Thus the value of 50 was chosen as the value to be used. No values were examined that were lower than 50 for the reason that at 40 there is again an object size that is frequently used, in the same manner as the objects of size 20 words.

The effects of the scheme were two-fold. Firstly, all large objects could be allocated in segment 0 up to the limits of the runs completed, which was 500,000 byte codes. Secondly, the time per tick was drastically reduced. A figure of 0.57 seconds per tick was achieved, an improvement by a factor of ten.

The major concern with implementing a paged memory system was that the fault rate would be so great that any improvement would be offset by the increased level of transference from disk. In practice this did not prove to be a problem. Data taken when producing the results for Table 4.5 gave an average free value of 6800 words free per disk segment used as paging area, which is 10% of the area used.

Because segment one was being removed as a block when the total amount of free space in it fell below the limit set for this to occur, memory compaction was no longer required. Because of removal of whole segments and because the virtual memory systems were later discarded in favour of wholly RAM based systems when more RAM became available, very little need was found for memory compaction (see Section 5.4) and it was not re-introduced at any stage.

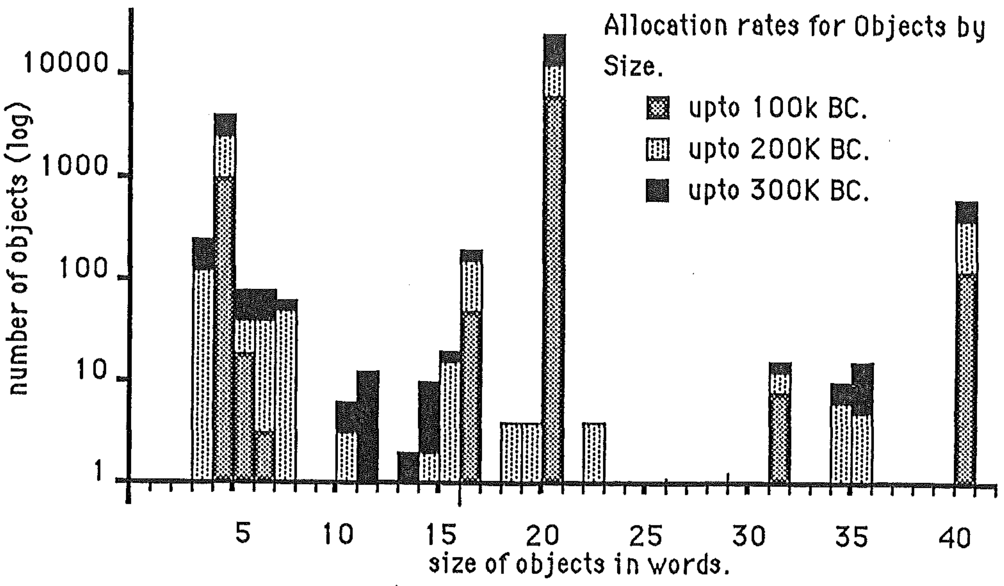

Figure 4.7: Allocation rate of objects over the first 100, 200 and 300,000 bytecodes.

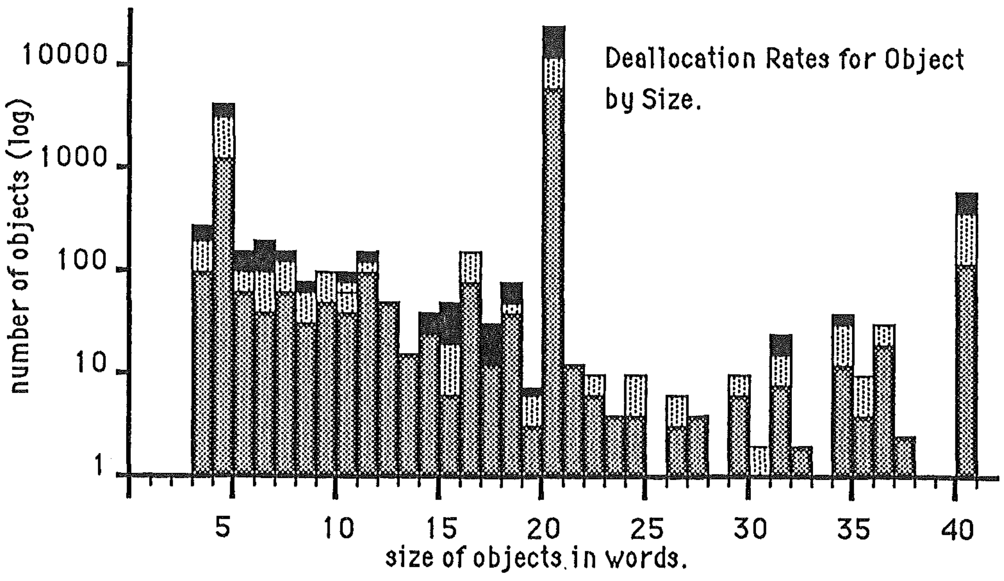

Figure 4.8: deallocation rate for objects over the first 100, 200 and 300,000 bytecodes.

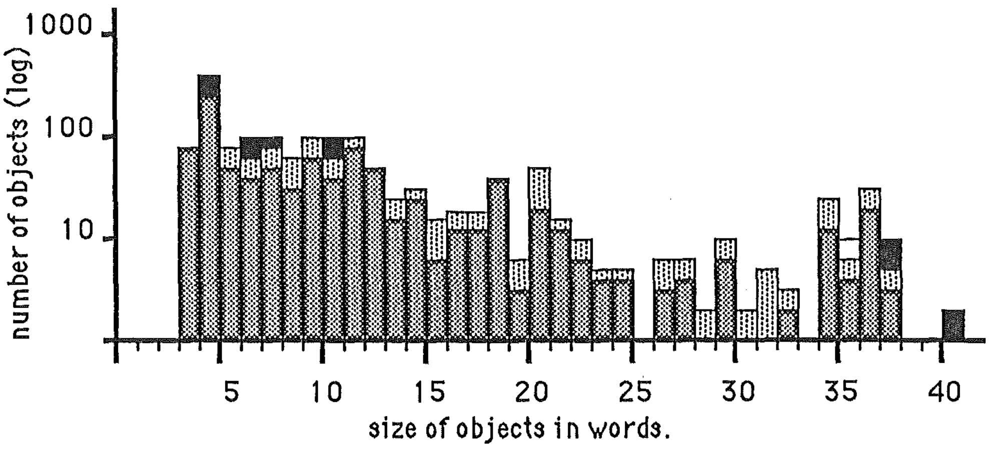

Figure 4.9: Fault rates for objects over the first 100, 200 and 300,000 bytecodes.

Chapter V

Organisation of Memory

5.0 Introduction.

One of the aims of this work was to investigate the effect that various memory management systems would have on the performance of the Smalltalk-80 system. Because of the nature of the virtual memory Smalltalk-80 system initially implemented, it was concluded that the method best suited to the examination of large numbers of different strategys would be simulation, this was because the physical mechanism involved in the implementation could be ignored.

In this chapter the results obtained from the simulations carried out to achieve this end are detailed. These results are then compared with the effects of implementing some of the methods examined on the Smalltalk-80 systems implemented, both in virtual memory and RAM only systems.

The following assumptions are used in all of the simulations that were conducted.

- That all memory was contained in one medium. It is treated as if it were contained completely in RAM.
- There is no compaction. As a consequence, it is possible for a run to fail because of lack of memory.

It should also be noted that though this chapter appears after the chapter on virtual memory, both sets of work were being conducted at the same time. Further, some of the work presented here was also being conducted at the same time as the first RAM only systems were being implemented (Sections 5.9 and 6.5).

5.1 Simulation

It was decided to perform the simulations in a trace driven manner. This was done as there was no reliable manner to establish a function that approximated the object allocation and deallocation distribution. The reason for this is that the majority of the allocation / deallocation activity (here-after termed transactions) occurs in the lower regions of the object size range, with notable specific peaks with objects sized 4, 20 and 40 words. Because these object sizes are dominant they must be accounted for. It is also not known what effects the order of allocation/deallocation has on the system and, as this cannot be built into any simple distribution function, traces were used as the raw driving data of the system.

The traces were obtained by including Pascal write statments in the Smalltalk-80 memory management code giving the following details,

1) the type of transaction - either an allocation or deallocation.

2) the size of the transaction in words.

3) the object pointer of the objects involved.

This last item of information is important when performing simulations that use a segmented address space. This is because objects that are allocated in one segment must also be deallocated in that same segment, and some mechanism is required to associate allocations with deallocations. However this requirement is not present in the linear memory systems, these systems are introduced in Section 5.9.

Because of the limited disk space available on the Lisa, it was decided to run the simulations on a VAX 11/750 under 4.2BSD Unix. This also allowed work to be carried out simultaneously on both systems.

To reduce the amount of work that had to be redone in gathering the data for each trace, the process of data collection was split into two phases. The first of these was to create a file containing a list of all objects allocated during the Virtual Image load phase of a run. This set of data represents the initial state of the system. As this set of data remains consistent for each and every run of the system, it needs to be collected only once and a special run was used to do this.

The collection of the remainder of the traces was done by including in the procedures **allocate** and **deallocate** write statements of the form described above. Because the collection of data from the virtual image load was done separately the load's completion occurred in the normal span of time.

The only factor governing the collection of data in the second phase is that all the data must be collected in one run because of the nature of the Smalltalk-80 system. This is due to the nature of the object allocation, as the strict order of transactions affects which object pointers are used for each transaction. Even while the system is "idle" the interpreter continues to allocate and deallocate objects. Though the system probably reaches a steady state during idle execution, using and reusing the same objects, there is no way that we can ensure that the same objects are used in any two separate runs.

The simulations were carried out over a set of three trace files, the first being collected from one of the virtual memory systems and the latter two from RAM based systems after the preliminary work had been completed. This reflects the relative performance levels of the two systems involved and the fact that the virtual memory systems used most of the free disk space for disk segments.

The trace file for the first of the RAM based set of traces,"Mscript" was obtained to check the results produced by using the file "s100" obtained from a virtual memory system. The third file was obtained to investigate some "unusuall" behaviour found when executing certain functions in the Smalltalk-80 system. Some of the details of the files used in the simulations are listed in table 5.1 below.

file name.	size in bytes.	number of transactions.	number of bytecodes of execution.	type.
iload	180,008	18,392	0	load of virtual image.
s100	177,791	16,432	100,000	first trace.
Mscript	2,034,370	208,976	1,000,000	second trace.
factM	382,391	41,427	1,000,000	third trace.

Table 5.1: Summary of files used to drive the simulations.

Pascal was used as the simulation language. It was chosen mainly because of the ease with which pointer structures such as linked lists may be manipulated and because of the author's familiarity with using it.

5.2 Details of the First Trace, s100.

This trace represents the execution of the first 100,000 bytecodes. Of these, the first 62,000 bytecodes are involved in the display of the first 3 windows of the system to become visible to the state shown in the manual that came with the tape [Xerox 1983b]. The windows are the:

- SystemBrowser,
- SystemTranscript and
- SystemWorkspace.

The remainder of the 100,000 byte codes executed are used to "print" the result of the string "1+2+3+4+5+6" in the lower pane of the SystemBrowser window. Specifically what is done is:

- the lower pane of the SystemBrowser is selected as the active window,
- the string is typed into the lower pane,
- the string is selected,
- the yellow button menu option "print it" is selected and activated and
- the string parsing and executing phase is then entered into.

The last stage given above does not run to completion as the whole run is terminated after 100,000 bytecodes have been executed.

5.3 Details of Mscript.

The first 100,000 bytecodes of this run are identical in function to what occurred in the run that produced the trace s100. The remaining 900,000 bytecodes were used to:

- print the result of executing the string "1+2+3+4+5+6"

- type the string "Form fromUser edit" on the same line as "1+2+3+4+5+6" and its result, and select it.
- select and activate the yellow button menu option "do it".
- execute the form editor for sufficient bytecodes to
    - select a region of the screen display and have it brought up on the screen along with the form editor control panel.
    - paint a section of the form selected black.
    - select the bit editor and a small section of the displayed form and use it to turn off a number of pixels in the section selected.
    - exit from the bit editor back to the form editor.
- select the SystemBrowser window without exiting from the form editor. This was done to keep the memory usage high.
- select and activate the "add category" option of the yellow button menu in the upper left pane of the SystemBrowser window and to add as an object class most of the object FinancialHistory described in Goldberg and Robson.

The run was completed before the FinancialHistory addition could be completed. The aim in collecting this trace was to try and obtain a more balanced selection of transactions that would be representative of a Smalltalk-80 system in actual use.

5.4 Details of factM.

The execution profile for the 1 million bytecodes of the factM trace follows the same format as s100. Here the string "1+2+3+4+5+6" has been replaced by "200 factorial" as the subject of the option "print it".

For storage of all significant figures, the calculation of the result of 200 factorial uses the class large positive integer. This class of object is capable of representing numbers with up to 131068 digits. As memory compact had been removed (section 4.15) when computing the results of the strings "1000 factorial" and "500 factorial", the Smalltalk-80 system was unable to allocate objects of size 160 and 181 words respectively. Examination of the dump of large free objects that was produced showed that large objects were being allocated in a size increasing manner, a section of one of the list of large objects is shown in table 5.2 below.

```
head of list -> 159, 110, 158, 158, 157, 156, 155, 156, 155, 154, ....
                45, 45, 44, 44, 43, 43, 42, 42, 42, 41, 41, end of list.
```

Table 5.2: The start and finish of the linked list of large free objects for "1000 factorial"

What has happened is that as the size of the large positive integer, that is the result of "N factorial", grows larger objects are required to hold the digits in the result. The effect of this is that a sequence of objects is allocated and deallocated producing a large degree of fragmentation in memory.

The calculation of 200 factorial was the largest calculation attempted that was successfully completed. Because of the nature of the transactions that occurred in this type of calculation, it was decided to use a 1 mega-bytecode trace, factM, as a further check on the performance of the memory management systems being simulated.

5.5 Simulation Aims.

By running these simulations it was hoped that a "good" method of organising the Smalltalk-80 memory could be located. What is inferred here by the term "good" is that the system:

- has the ability to allocate objects with the minimum of searching and hence execute quickly and
- does not fragment the memory space to too great a degree.

The first of these aims can be met by comparing the number of nodes that are examined when allocating objects. If it is assumed that there is a set minimum of work that must be completed with examinations during a search then a method that makes N comparisons will execute more quickly than one that makes N + M.

The aim of the second constraint is to try and ensure that large objects can be allocated quickly. If the free memory is split into a large number of small units, it is probable that if sufficient space exists in the system for allocation to proceed a compaction phase would have to be entered into before the actual allocation took place. If this is the case, and the allocation of large objects occurs frequently, the time taken for compaction during allocation will tend to dominate the run time. This is the case in computing the values of large factorials.

What is required then is a method that keeps fragmentation of memory to a minimum, that is keeps objects of large size available. To gain the required information from the simulations run, the following data was gathered.

- The total number of free nodes examined to find space for the allocation of objects(nodes seen), excluding the Image load section of the run.
- The number of nodes free at the end of a simulation.
- The mean size of the free objects at the end of a run.
- The size of the largest free object remaining in the memory space at the end of a run.

The first of these figures gives the indication of the execution speed of a system. The second and third give an indication of the fragmentation that can be expected to occur, and the last figure indicates the system's ability to allocate large objects.

The total number of free nodes examined is a good indicator of the speed that objects will be allocated because of the following:

- most of the code involved in allocating objects is used in the splitting of objects.
- all methods differ only slightly in the complexity of the remaining code required to implement them. The most complex sets of code are assocated for the segmented methods of allocation.

5.6 System Classes.

The memory systems examined fall into two major classes: segmented and linear systems. Each of these two classes can be further broken down into systems that use at most two lists and those that uses a larger number lists. The two list systems use one list for objects larger than some limit N and another for objects that are the same size or smaller. In multiple list systems all but one of the lists are used for only one size of object; the last list is used in the same manner as the list of large objects in the two list system. The multiple list class includes the free list structure described in Section 2.5 as a sub-member, these schemes use multiple lists for the object sizes at and below the limit set.

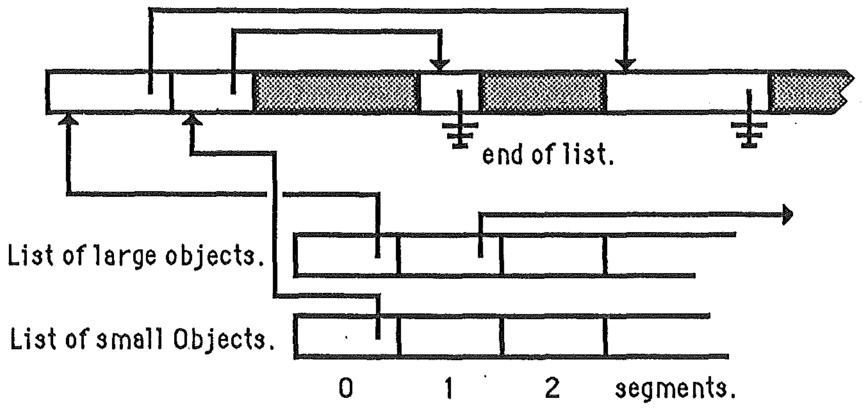

Figure 5.1: Example of free list structure with two lists per segment.

The allocation method used for the majority of systems examined in this section is first-fit [Knuth 1973a]. That is, the allocation of an object occurs using the first suitable space found. When objects are deallocated they are placed at the start of the correct linked list of free objects. When an object is split to make two smaller objects, the section remaining that is not allocated is either moved to the start of the correct list for its size or left in place on the list that it is currently on.

5.7 Segmented Memory, Twin Lists.

These systems are based on the organisation described in section 2.5. Memory is split into a number of 64k word objects that are initially free. Objects are allocated either by splitting a larger object into two smaller ones, one of which is the size requested, or by allocating an existing object of the correct size. If allocation occurs by the second of these methods, then the object is simply removed from the linked list of which it was a part. If an object has to be split to provide an object of the correct size, then it is treated in the manner described at the end of Section 5.6.

All simulations discussed in this section were made using six memory segments.

The first system examined used only one linked list per segment. The results are shown in Table 5.3 below.

	s100	Mscript	factM
nodes seen	273,769	9,242,716	3,132,514
nodes free	2,498	12,055	3,873
mean node size	56	8	27
max. node size	39,306	---	27

Table 5.3: Results of simulations for a one list / segment system, with 6 segments. The run using the trace file Mscript failed to complete as it could not allocate an object because of memory fragmentation.

As can be seen, all simulations required several million comparisons to achieve their results. Most important is the fact that the simulation using Mscript failed to run to completion. As would be expected, there is a high level of fragmentation inherent in this system, as the mean size of the free objects in the run utilizing Mscript indicates.

The results of the simulations conducted with two free lists per segment are shown in Figures 5.2 to 5.5. The graphs are drawn with the X-axis showing the value where the split into two lists occurred; plotted points on the X-axis are marked by labels. For example, in Figure 5.2 the plotted points occur at 5, 10, 13, 15, 18, 19, 20 and 21. The Y-axis shows the relationship between the values for different values of N for different runs. Values on the y-axis are plotted as a percentage of the largest value for each run. For Figures 5.6 to 5.11 the value plotted on the X-axis is the size of the largest H-list used. The complete set of data from the simulations is given in Appendix 4.

The first of the graphs, Figure 5.2, shows the number of nodes examined in each run. The first factor to be noted is that the split at N = 20 and N = 21 produces the worst results shown (it has the largest value). Also, like the one list per segment system, this system failed to complete the N = 20 using Mscript. Results from that run are included because they show what could be expected if the run had completed.

Figures for the data collected show a tendency to either increase or decrease as the value of N increases. For instance, there is an increase of execution speed (predicted by value of the number of nodes examined) from N = 5 to N = 19 as shown in Figure 5.2. Likewise, there is an increase in the size of the largest free object available over the same range, as can be seen from the data gathered with runs using the trace "Mscript". This is shown in Figure 5.5.

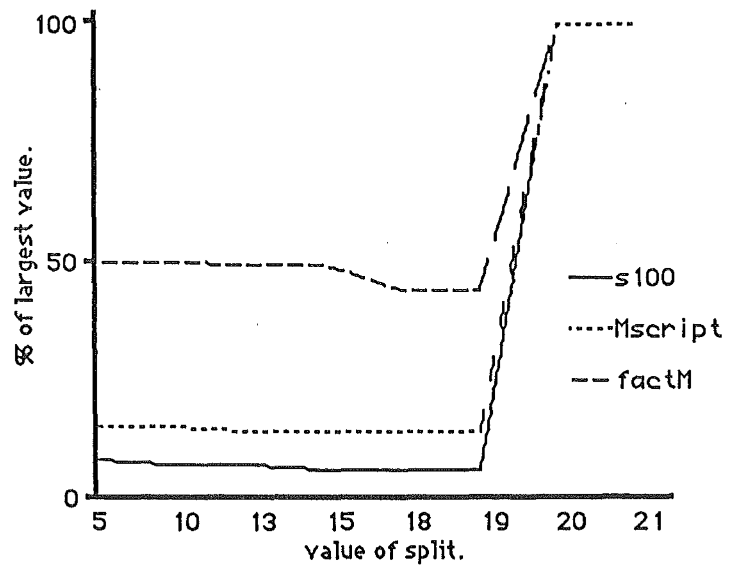

Figure 5.2: Number of nodes examined in simulations using a two lists per segment system, with 6 segments. Note the trace "Mscript" with N = 20 did not complete.

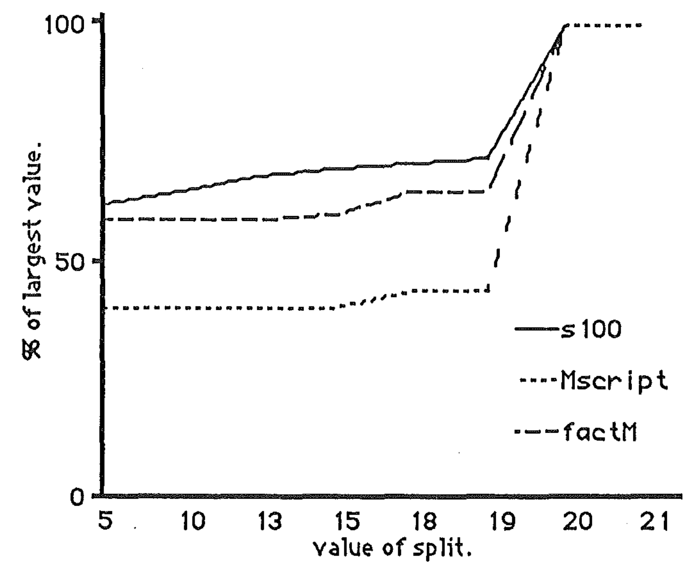

Figure 5.3: Number of nodes free at the end of the two lists per segment simulations.

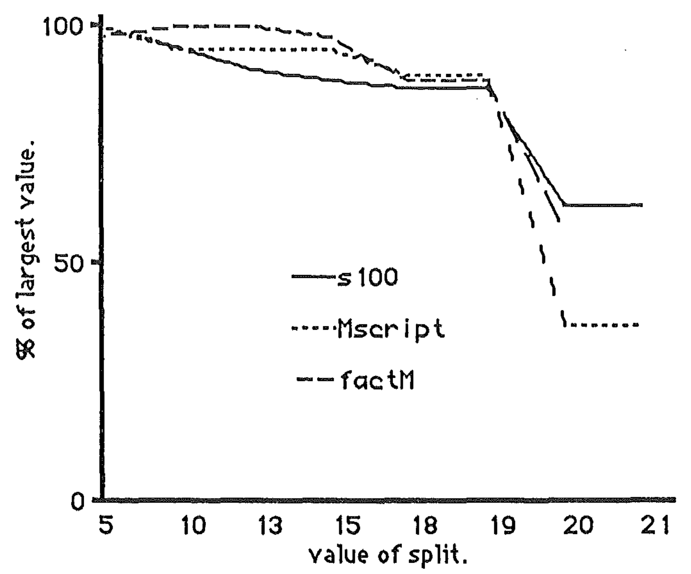

Figure 5.4: Average size of the nodes still free at the end of the two lists per segment simulations.

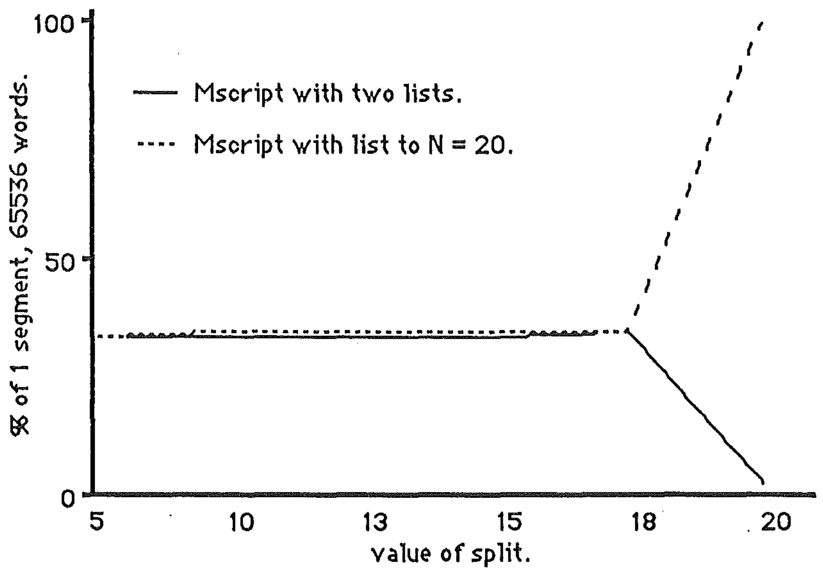

Figure 5.5: The size of the largest object remaining in the system as a percentage of one segment. The solid line is from two list systems. The dashed line is from a multiple list system.

The data for the number of nodes free (Figure 5.3) shows an increase across the table, except for the data collected from the trace factM. A similar pattern is shown in the data collected on the average size of free objects. The data from the traces s100 and scriptM shows a constant decrease in the average size of the objects as N increases.

When comparing the results from these simulations with the information obtained from the implemented system, detailed in section 4.14, the speed increase that would be expected from the reduction of the number of comparisons done, is present. However the results do not explain why the peak rate of execution is seen at N = 13 in table 4.3.

The sharp rise in the number of nodes examined, the number of nodes free at the end of the simulations and the drop in the average size of the free objects shown in Figures 5.2 to 5.4 at a split value of 20 are a direct result of the way objects of this size (20) are used. It was seen in Figures 4.7 and 4.8 that a very large number of objects of this size are being allocated and deallocated. When an object of size 20 is deallocated it is placed on the front of the list of small objects. If an object of a size less than 20 is allocated before the object can be reused without splitting, then the object will be split. The results of this are that a longer search has to be made of the list of small objects and that large objects from the other list have to be split more often to provide new objects of size 20. This does not occur when objects of size 20 are on the large list, as they will be split infrequently.

5.8 Segmented Memory, Multiple Lists.

Cohen [1981] in his survey of garbage collection methods raises the issue of delaying the compaction of varisized cells for as long as possible. The procedure suggested for doing this is to have a number of free-lists. Of the N lists available, N-1 of them would be used to form lists of one size of cells that are commonly used. These lists are termed homogeneous lists, or H-lists. The remaining free list is used to form a free list of cells of miscellaneous sizes, this list being termed the M-list.

When multiple lists for objects below and at the limit N are introduced, the trends are much more conclusive. As shown in the Figures 5.6 and 5.7, there is a steady improvement in the figures chosen to measure performance with increasing values of N, the largest change coming when N is increased from 18 to 20.

The algorithm used to decide which segment to search for a free object, used in producing the results so far, is given below.

```
readln(infile, kind, size, objectName);      { get data from trace file }
seen := 0;                                   { no segments seen }
seg := CurrentSegment;
while (not found) and (seen <= 6) do
begin
    found := FFscan(seg, big, size);         { scan the M-list to get object }
    if not found then
    begin
        seg := (seg+1) mod 6;                { not found goto next segment }
        seen := seen + 1;                    { number of seg's seen }
    end
    else                                     { have found object }
    begin
        outof[objectName] := seg;            { note where objects comes from}
        checkSum : checkSum - size;
    end;
end;
CurrentSegment := seg;
```

Algorithm 5.1: The segment changing scheme from Goldberg and Robson as used in the simulation of the one list system.

This is the method given in Goldberg and Robson [1982c]. A set of simulation runs was done using a slight modification to this system, as shown in Algorithm 5.2 below:

```
seg := 0;                                    { change made here }
while (not found) and (seen <= 6) do
begin
    { the same as above }
end;
```

Algorithm 5.2: The modification made to segment changing scheme.

As can be seen from the data given in Appendix 4, the results show that there is no significant difference in the effects of the two methods presented. The second algorithm however has the advantage of only requiring local storage, whereas Algorithm 5.1 requires the use of a global variable. The importance of this is that, in theory, access to local variables is faster than accessing global variables [Forsman 1983]. In practice no effect was observed when using a local variable in place of the global variable CurrentSegment.

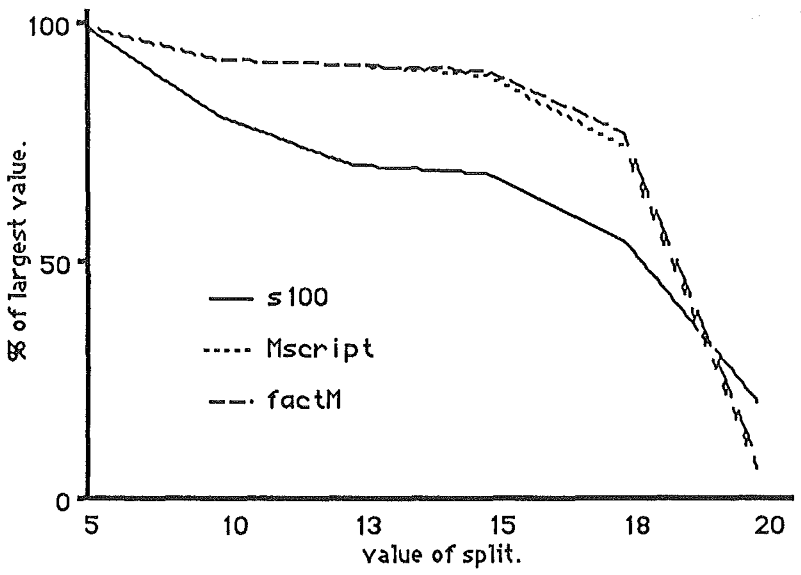

Figure 5.6: The number of comparisons used in the simulation of a multiple list segmented memory.

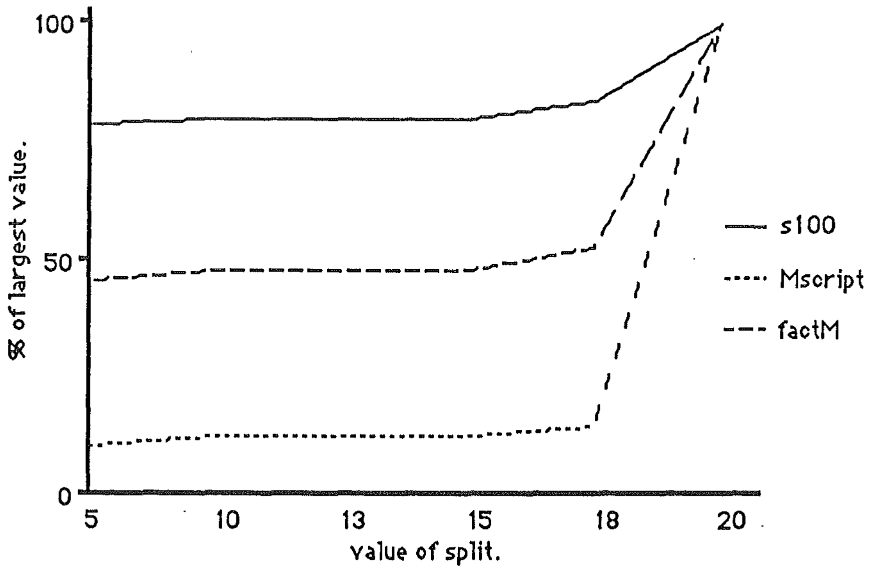

Figure 5.7: The average size of free objects remaining at the end of simulation runs using segmented memory with multiple free lists.

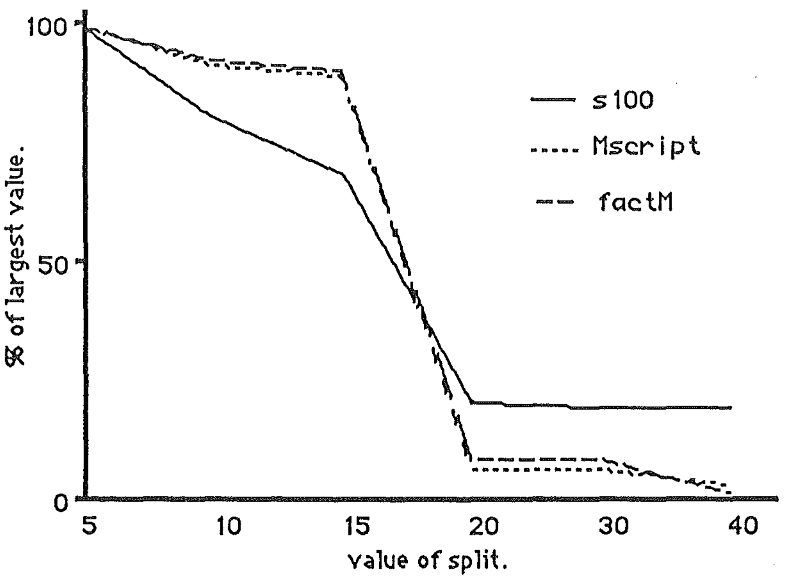

Figure 5.8: The number of comparisons done in simulations using segmented memory with multiple lists and Algorithm 5.2.

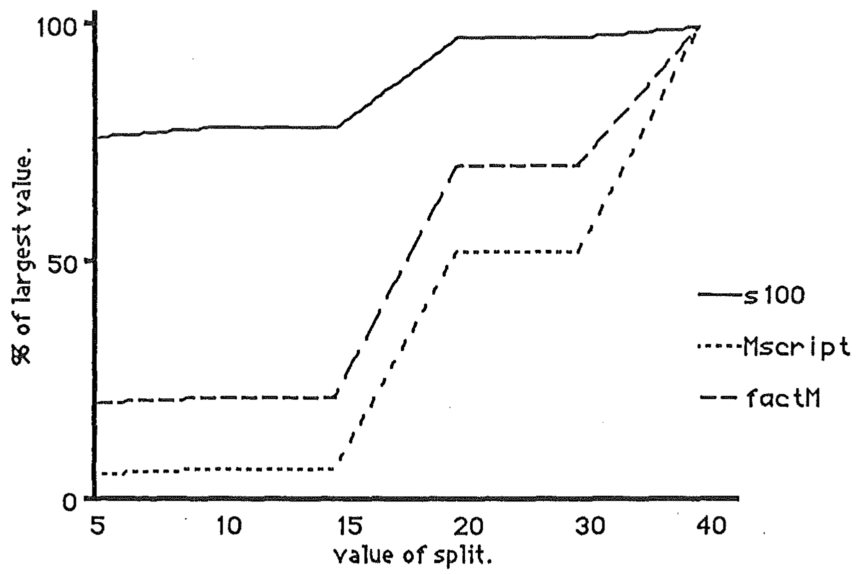

Figure 5.9: The average size of free objects at the end of simulations using segmented memory with multiple lists and Algorithm 5.2.

The set of simulations that were run using Algorithm 5.1 were extended so that the H-lists were used up to a value of 40. The most important information to be extracted from the results of this modification is that there is a continued improvement up to the list limit of N = 40, as shown in Figures 5.8 and 5.9. The effects of this are most notable when taking into account that data on the average size of node remaining. For the two 1 million bytecode traces a large improvement is shown. The results form simulations using "Mscript", with N = 40 being nearly twice as "good" as the runs with N = 20 and N = 30 (Figure 5.9).

5.9 Linear Memory

The manner in which the Smalltalk-80 address space is organised dates back to the Xerox Alto computer designed in 1973. This had an address space of 64k 16 bit words. This address space could be increased by adding further banks of 64k words up to a maximum of 256k words, or 4 segments [Thacker 1982].

The system used here to implement Smalltalk-80 was based on the MC68000 chip. This chip has a linear address spaces of 16 megabytes. This allows memory to be used in a linear fashion using one 32 bit word to store the complete location of an object. Because a linear memory management scheme fits the instruction set more closely than does the system used in Goldberg and Robson, simulations using such a scheme were run. The results for the number of comparisons and mean object sizes are given in Figures 5.10 and 5.11.

The data used in these graphs was produced using a set of 6 segments and, when splitting an object, placing the section that is not allocated at the start of the correct list for its size. At the start of a run using this system, when all pointers are either unused or point to free objects, the list of large objects is a linked list of 64k word (16 bit) free objects. The limit of 64k words as the maximum possible size of objects has been kept because it:

- results in the minimum change to the system code if introduced in to an actual Smalltalk-80 system,
- saves space, allowing objects of a size that requires 32 bits to store, would cause each object to have 2 extra bytes associated with it and
- enables simple comparisons with previous results.

The splitting in the linear memory systems was altered slightly from the scheme used in the segmented memory simulations. In this system the unused sections of a split object are always moved to the start of the list on which they should be on, even if they are already part of that list.

When making the comparisons between the results from the simulations using segmented and linear memory with multiple lists, it can be seen that the figures for the number of nodes visited are very similar, for the trace s100 there being a difference of around 200-500 in the upper three values of N. The number of comparisons done with N = 40 for Mscript differs by only 2%, so the methods can be considered to be equivalent in this respect (Appendex 4). This is despite the fact that the data from the segmented systems (figures 5.8, 5.9) is obtained from a system where there are inherently more comparisons, due to the manner in which the unused portions of split objects are handled.

To see if the manner in which the split was done in the segmented memory simulations was significant, a further set of runs were undertaken with splitting being performed in the manner used for linear memory. The results of the runs are given in Table 5.4.

split at N	20	30	40
nodes seen	107,006	107,000	106,608
nodes at end	626	705	338
mean size	167	145	309
max. size	65,536	65,536	65,536

Table 5.4: Results using the split object placement method used for linear memory, with segmented memory. The trace file used is Mscript.

As can be seen, the number of comparisons required for N = 20 and N = 30 in the segmented systems are lower. However the number of comparisons needed at N = 40 is still some 2% higher in the segmented system.

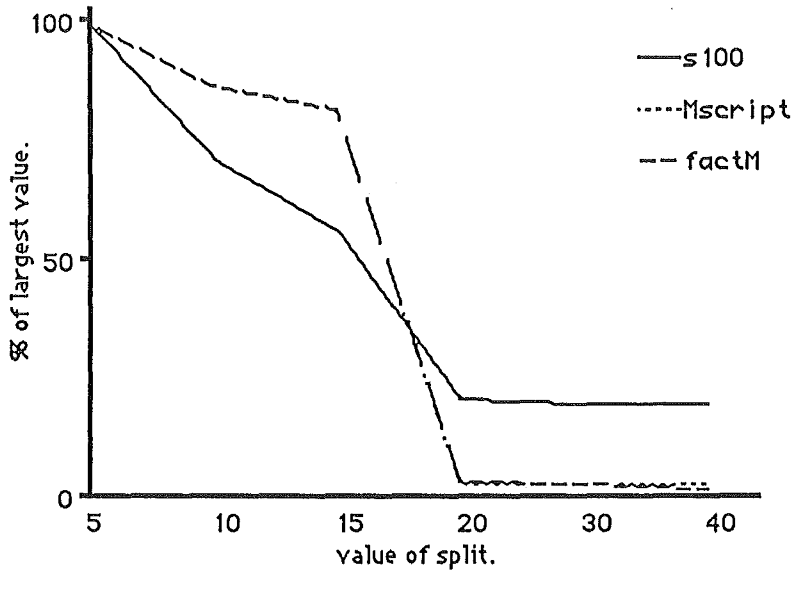

Figure 5.10: The number of comparisons done in simulations using linear memory with multiple lists.

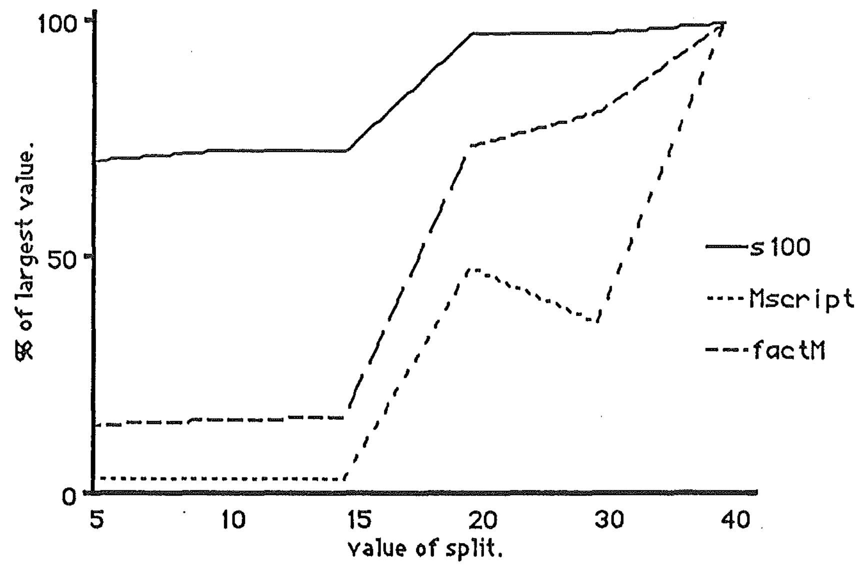

Figure 5.11: The average size of free objects at the end of simulations using linear memory with multiple lists.

The most important point to note is the difference in the average size of free objects left in each of the systems. In Table 5.4 the average size of the free objects for linear memory run with N = 40 is twice that of the segmented system, when using "Mscript" as the trace.

The fact that the linear memory system is better at keeping fragmentation low is highlighted by the data in Table 5.5. This data is for "Mscript" driven runs with N = 40 when using only five segments.

	Segmented	Linear
nodes seen	106608	106463
nodes at end	337	165
mean size	115	236
max. size	22232	30777

Table 5.5: The effect of having only 5 segments for allocation.

The data shows that there is a large difference in the size of the largest object available. The largest object in the linear system is 27% larger than the largest object in the segmented system. This size difference also accounts for 16% of the space available in the largest possible 64k word object.

To determine if there was any further gain in having individual lists above the value of N = 40, simulations using all values up to the value of N =100 were carried out. The results showed that there was no gain of any significance in having lists for single sizes above 40.

5.10 Comparisons With Actual Systems; Validation.

Results obtained from simulations should be verified in some manner. In this section the effects of some of the methods simulated are compared with changes made in the Smalltalk-80 systems implemented.

The change from using one list per segment to using two lists per segment is detailed in Section 4.14. The change from a segmented memory system to a linear memory system was made early in January 1986. The change involved

- altering the way in which the location bits were accessed, as they were being expanded from 16 to 32 bits and
- removing the access to the segment bits and their addition to the offset in the segment.

The result of this change was an increase in the rate of bytecode execution of 9%. However this was probably due to the reduction of work that was done on each object access rather than an indication that the linear memory system was intrinsically better than the system using segments.

A more useful test was completed on a linear memory system by performing a series of runs using the implemented RAM only Smalltalk-80 system. The results of these runs are shown in Table 5.6.

split value N.	time (seconds)	nodes examined.	pointers free.
5	575	1,090,608	9696
10	532	813,539	10060
15	460	365,102	10069
20	405	43,679	14259
25	404	43,678	14259
30	405	43,678	14259
35	404	43,623	14286
40	405	43,609	14304

Table 5.6: Results of using different cut-off points **N** in an implemented Smalltalk-80 system

It can be seen that Table 5.6 follows the same pattern as Figure 5.10, although here there is no definite indication that N = 40 is a better cutoff point than N = 20. What is shown is:

- results from using values of N greater than 10 show dramatically better results. Split points at 20 and above are a factor of 20 better than the value of 10 and
- for systems with H-lists at values greater than 20 we have a small but constant improvement in the performance. It is most notable in the number of free pointers in the system. The system with N = 40 being 45 times better than N = 20 and 18 times better than N = 35.

As a further check on the simulations a series of run were completed using the trace "Mscript" as the driver. The statistics gathered in these simulations were collected after the occurrence of every 20,000 allocations; deallocations were ignored. This was to get a series of runs that used the same data to examine the degree to which the system had stabilised. This was done as a further check on the validity of the simulations. The results are shown below in Table 5.7.

	number of allocations used for stats.					
split N	20,000	40,000	60,000	80,000	100,000	To End.
5	100.0	100.0	100.0	100.0	100.0	100.0
10	86.2	86.4	86.7	86.9	87.0	86.7
15	80.3	81.6	81.6	82.1	82.3	82.0
20	8.8	5.4	3.8	3.9	3.4	3.2
30	8.8	5.4	3.9	3.9	3.4	3.2
40	8.6	5.3	3.7	3.8	3.3	3.1

Table 5.7: Results for comparisons as a percentage of highest value at various stages of simulations using the trace file Mscript for various values of N.

It can be seen that the values produced for the small values of N are all very similar and can be considered to be stable, however the values for the upper values of N on the table show a large initial change as the length of the simulation run becomes larger (left to right across the bottom three rows). The rate of change does slow on the right side of the table, from 3.3 % to only 0.2 %. This again can be considered to have stabilised, as the nature of what is being done with the system has also changed across the table and it may have some effect on the results, noting that, at this end of the simulation, the system is interacting with the user typing and spending more time than other sections in the wait state.

The conclusion that is reached is that the simulations appear to be an adequate representation of the way that an actual system will behave.

5.11 Other Systems.

Two other methods of organising memory were examined briefly to see if they offered any further performance benefits. The first of these was a modification to the 41 list linear memory system. It was noted in Section 5.5 that one of the factors used to characterise a "good" system was the level of fragmentation that occurs. It was decided that a system that contained free N objects would be better than a system that contained N + M free objects. However, this may not always be the case. We can examine two possible ways in which our assumption could be wrong.

For example, consider the case where there is an object of size 80 words at the start of the list of large objects. If no large objects of a size 80 words or below are allocated, then at each allocation of a large object, one extra node must be examined before the allocation can be completed. It would appear that more utility could be gained from the object if it were to be split into two objects of 40 words, or four objects of 20 words, these being the most commonly required largish object sizes. This situation is however unlikely to exist for any extended period, because there would be requests for small objects (3, 4 or more words) that would whittle the objects size down slowly until it "fell off" the front of the large list onto one of the H-lists.

For a second example, consider the case where a large object has been split near the end of the M-list. If the section *not* allocated is not small enough to be moved to one of the lists of single sized objects, but at the same time is relatively "small" in size, again say 80 words, then because the object is not at the start of the list, we must split any smaller objects that may be required for some other object in front of it. If the object that is split from is a very large object (several kilowords), then the situation arises where the size of the largest free object in the system is possibly being reduced.

There is not much that can be done about the first case. As was seen in Section 5.4 where the calculation of factorials was examined, very long lists of free objects can be formed in some cases. Using a method such as best fit [Knuth 1973a] could produce very long searches of memory in this type of situation.

The solution to the second case brings about the first case as its solution. If it is desired to make sure that all objects split are the objects "best" suited to being split, thenit is necessary to make sure that the smallest objects are at the start of the list of large free objects. To do this the list must be sorted after each split occurs.

The list of large objects is ordered to start with, in that all objects on it are the same size, ie 64k words. With each split, either the object that remains is moved off the list, if this is possible, or it is bubbled toward the start of the list, so that the list remains sorted. When adding an object to the list the same procedure is followed in reverse. That is, objects are added to the start of the list and moved toward the end. Adding objects at the end of the list is intuitively counter productive, as this is the point where the largest of the large objects reside. Experience to date shows that objects that are allocated then deallocated are not that large, so it is probable that there would be more movement required to sort an object down the list than to sort an object up the list.

	linear memory: unsorted.	linear memory: sorted.
nodes seen.	106463	105125
	21451	20904
sort comparisons.	0	170
	0	51
nodes free.	165	154
	284	283
mean object size.	236	253
	140	140
max. object size.	30777	30801
	34452	34452

Table 5.8: Comparison of sorted and unsorted lists of large objects for simulations using five segment equivalent memory space. For the traces Mscript (top) and factM.

Table 5.8 shows the results of the simulations carried out using a sorted list of large objects. It can be seen that the sorting of the lists has resulted in a reduction of the number of nodes that are examined when locating objects to allocate. We have gained a reduction of 1% in the Mscript and 2% for factM in the total number of nodes examined, including the examination of nodes during sorting. There has also been a small reduction in the level of fragmentation produced when running Mscript, as well as a small gain in the size of the largest object left in the system at the completion of the run. The same figures for factM however show no change. This can be expected as there is but a single object left on the list of large objects at the end of the run.

The second system to be examined was the buddy system [Knuth 1973b]. It should have been noted that the smallest object that may be allocated has a size of two words. The largest object that may be allocated is 64k words, which is one segment. As these are both powers of 2, they map neatly onto the object sizes used in this system. A simulation using Mscript was conducted using the buddy system. The results were, number of comparisons required 106,389, more than either the linear system with N = 40 or the segmented system. The number of nodes free at the end of a run was only 90 and the average size of free objects was 401 words, this being better than any of the systems described so far. However the largest object remaining after 1 million bytecodes was only 16384 words long (2 ** 14). The buddy system however required the equivalent of 7 segments to obtain this result. The other systems examined that were considered to be good required the equivalent of only five segments to achieve comparable performance.

The buddy system can be seen to offer no advantages over any of the other systems previously examined. The technique of sorting the free list of objects greater than 40 words however may have some benefit if some method can be found of "hiding" the overhead involved in sorting. The possibility of doing this is examined further in Section 7.4.

5.12 A Brief Analysis of "Mscript"

This section examines some of the features of the trace file "Mscript". The reason for doing this is the statement by Wirfs-Brock [1982b]:

> "A memory manager might choose to dedicate a memory segment to the allocation of contexts."

Contexts are the equivalent of stack frames and are created when messages are sent (procedures are called) to objects. They contain information such as local variables that do not have their values saved. Physically they are objects of either 20 or 40 words.

On examining the number of objects allocated of each size in the trace file, it was found that certain sizes of objects dominate allocation and deallocation transactions as shown in Table 5.9. As can be seen, most of the activity occurs with objects sized 20 words, and this explains why the systems examined that have an H-list for objects of this size are so much better than other systems.

size or range	% of allocations	% of deallocations.
4	15.6	14.7
20	68.5	80.5
2-10 not 4	9.3	1.9
	======	======
	93.4	97.1

Table 5.9 : The percentage of allocation / deallocation activity of certain sizes of objects.

In light of the data presented, it would seem that a separate segment for contexts of size 20 (contexts of size 40 account for only 1.8 % of transactions) could produce a significant increase in performance.

To check the validity of this, it was decided to determine the maximum number of any object that were present (allocated) at any one point in the trace file. The results of this investigation are given in table 5.10.

size of nodes (words).	nodes from "iload".	max. number of nodes.	space(words) requires.
4	3095	3,836	15,344
20	233	427	8,540

Table 5.10: Maximum numbers of objects allocated for the sizes given using the trace files "iload" and "Mscript" and the space occupied.

From this data, it appears that if a separate segment was to be provided for objects of a single size then objects of size 4 would be better, because of the larger space that they require. To check these results a series of runs were made using the Smalltalk-80 system implemented, with hooks added to the allocation and deallocation routines to gain similar information.

type of run	max. number of words of size = 4	max. number of words of size =20
drawing a mandala	81	77
printing "200 factorial"	17	288
using Form Editor	512	151
browsing (1)	15	80
browsing (2)	92	143
adding Class FinancialHistory	80	133

Table 5.11: Results gained from running the Smalltalk-80 system of the maximum numbers of objects sized 4 and 20 words. Data does not include load phase.

What is not considered in the above discussion is that linear memory systems appear to use memory more effectively. Thus the idea of a "segment" is redundant. However the segment concept can still be applied to any arbitrary area in memory and may be useful for memory compaction. The aim would be to create an area that contains a large number of "fixed" objects and free objects (such as contexts) that are used frequently and need not be re-examined.

Chapter VI

Space Reclamation

6.0 Introduction.

This chapter covers the area that has the most effect on the performance of the Smalltalk-80 system after the logical organisation of memory. That is the identification and removal of objects that are no longer performing a useful function in the system. Garbage collection in the Smalltalk-80 system is usually based on the use of reference counts. For every pointer to an object that is created, the count field in the object's ROT entry is incremented by one. For every pointer to the object that is destroyed, the count field in the objects ROT entry is reduced by one.

When the value of an object's count becomes zero, that object has no references to it and *should* be deallocated so that the space that it occupies can be re-used. Any objects that are referenced from an object whose count is equal to zero *must* have their own count bits decremented by one as they are effectively no longer referenced from that object.

The procedure for decreasing an object's reference count is vital to the performance of any Smalltalk-80 system. Measurements show that, on average, there are two calls to the procedure **storePointer**, which in turn calls the procedure **countDown**, associated with the execution of each bytecode. This figure does not include the explicit calls to **countDown** made via an interface procedure from interpreter routines.

The purpose of this chapter is to examine some of the methods that can be used to perform the function of "counting down" an object's reference count and comparing the performance and merits of some of the various systems described.

Figure 6.1 below shows a set of secondary references to objects from an object (333) that will be taken to be the root object. This example will be used in showing how some of the methods described perform their various functions. The objects numbered 666, 777 and

888 will be treated as if their count fields are zero after the pointer from object 333 has been removed. That is, they have only one pointer to them from object 333. Objects 17 and 13 will be treated as if they have counts that have reached the upper limit and cannot be reduced. This is in fact the case in the actual system where the count values for the object pointers 0 to 26 have a value of 255.

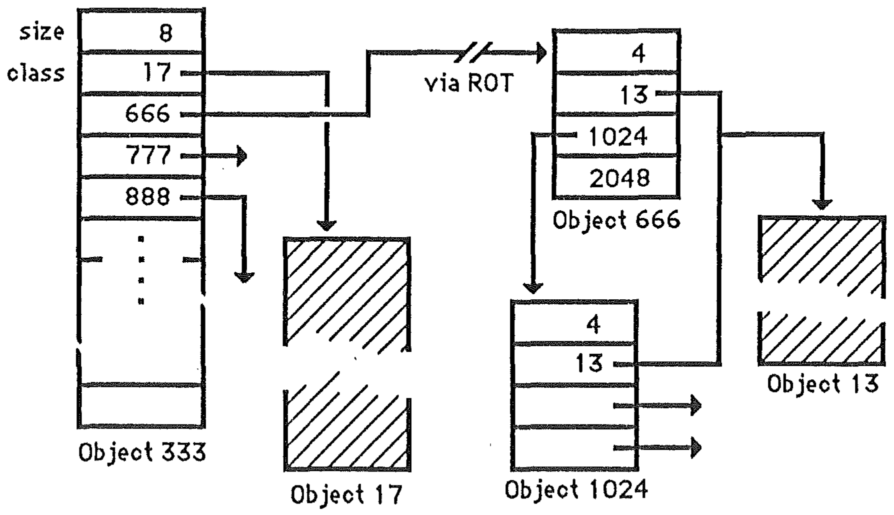

Figure 6.1: The example network of objects, to be used in describing the behaviour of various count reduction methods.

6.1 Recursion Again .

When an object's reference count reaches zero the object must be deallocated. If the object contains pointers to other objects then their counts must also be decremented. If this further decrementing reduces another object's reference count to zero then that object must also be deallocated. The problem and function definition are inherently recursive. Also there is no means of predicting the depth of search that will occur (the longest chain of pointers that exists). Krasner and McCullough [1984] state that this method of recursively decrementing the counts of objects "...may use a tremendous amount of interpreter stack space".

To overcome this, counting down of an object's references was done as a two stage process. The first stage was to reduce the count of the root object. If this object was then to be deallocated, any objects that it pointed to had their count bits decremented. However, unlike the root object, these secondary objects were not deallocated when their count bits reached zero. They were left allocated, with their counts at zero, as "junk" in the system.

The second step in this process occurred when the amount of "junk" left in the system exceeded some limiting value. A linear scan was then made of all object pointers looking for objects that were allocated but had counts of zero. These objects were then deallocated by calling the procedure **countDown** with them as root objects. This process was repeated until all secondary objects were removed. The function performing this scan was called **seagull**.

The main side effect of using this method is that it wastes some space and ties up some object pointers for some period of time. This was considered a small problem when compared to the possibility of running out of stack space.

This problem is mentioned in Goldberg and Robson [1982b] where they state:

"To guard against stack overflow, the depth of the stack must be greater than the longest chain of pointers in memory. This requirement is difficult to satisfy when memory space is limited."

To try to avoid this problem, Goldberg et. al. introduced a method where the chain of objects that were to be deallocated was constructed using the the object's size bits to hold a pointer to another object. The size bits in turn were to be stored in the count bits field of the ROT. If an object was of a size greater than 255 words, the count bits field would be insufficent to store the size of the object. Their solution to this was to allocate an extra word at the end of the object to store the size bits if the object's size could not be stored in the count field (method 6.1.2). As an alternative method had been found to reform deallocation (method 6.1.3), this extra word was not required, the trade-off between time and space being made in favour of space, which initially appeared to be the greater problem.

An important side effect of not using either of the methods put forward in Goldberg and Robson was that the code for allocating an object was simplified, as the extra word at the end of a large object was no longer required to hold the size of the object.

The methods discussed so far will be referred to by the numbers given below,

6.1.1 To recursively call the procedure **countDown** for each new object that is to be deallocated (freed), using the interpreter stack space to maintain the chain of objects.

6.1.2 To explicitly form the stack using the objects themselves to hold backward links to where they were being deallocated from.

6.1.3 Leaving secondary objects (such as object 666 from figure 6.1) "allocated" in memory but with a reference count of 0 and at some later stage doing a linear scan of the Resident Object Table (ROT) to locate these objects and again perform the **countDown** operation upon them.

Each of these methods has advantages and disadvantages, some of the more important being outlined below.

6.1.1 + conceptually simple, simple to implement.
- it is possible to run out of stack space.

6.1.2 + is guaranteed to have enough space to form the stack.
- complex to implement. An extra word must be added to each object for the objects that are greater than 255 words in size. This complexity could be reduced by adding an extra word to every object, making the method more uniform; this would of course waste even more space.
- uses more space, at least 1 word for each large object.

6.1.3 + is simple to implement and understand.
+ requires no additional memory to run.
- requires more time to run than any of the other methods given because of the repeated scanning of the ROT.

6.2 Linear Scanning.

The linear scan (6.1.3) in its simplest form starts searching for objects that have been left allocated with zero counts at the beginning of the ROT and moves though it in a single pass until the end is reached. After the completion of this pass, all of the *initial* objects (those that existed before the pass started) that had been left in an ambiguous allocation state will have been found and deallocated.

The first consideration in implementing this method was when to schedule such a pass. It was decided that the best point to do so was just before the allocation of objects, thus allowing for nearly the maximum number of objects to be available for allocation.

Because of the large amount of time that each pass through the ROT would take and the large number of objects that are allocated (on a bytecode by bytecode basis), it was decided that some limit on the number of passes that were performed had to be made. This was done by keeping a count of the number of secondary objects that had been reduced to a zero count state. When this count went above a limiting value, a pass though the ROT was allowed to proceed.

The amount of time taken for a scan was further restricted by noting the value of the largest object pointer to be used at any stage. As the entriess in the ROT above this value do not contain any useful information, scans of the ROT can be halted at this point. The scanning of the ROT was further restricted by decrementing the count of the number of objects to be located as objects were found and removed. When the count of secondary objects yet to freed reached some lower limiting bound the scan was again terminated.

In method 6.1.3's final form the number of scans was even further restricted by only calling the procedure **seagull** to do a scan when allocating a pointer object. This was done because the vast majority of calls associated with the allocation of objects are for the allocation of pointer objects. A reduction of 6.4 % in the total time taken for the benchmark runs was gained from just this one change.

6.3 Object Linking.

Krasner and McCullough [1985] describe a method for freeing objects that is non-recursive and uses fewer resources than any of the methods so far discused. It is similar to method 6.1.2 but requires, on a per object basis, only one pointer and no extra storage. By this it is meant that no stack is required as in method 6.1.1. Also, the extra word of method 6.1.2 is not required because there is no backward pointer.

The procedure operates by linking secondary objects to be freed onto a list built by using the class field of these objects. The link is formed after saving the class field in a local variable and counting this value down immediately, the class field itself being a pointer to an object.

The only object types that are entered on the queue in this manner are objects that either have their pointer bit set (pointer objects) or are objects of the class type *method*, all other non-pointer objects having their class bits as their only pointer.

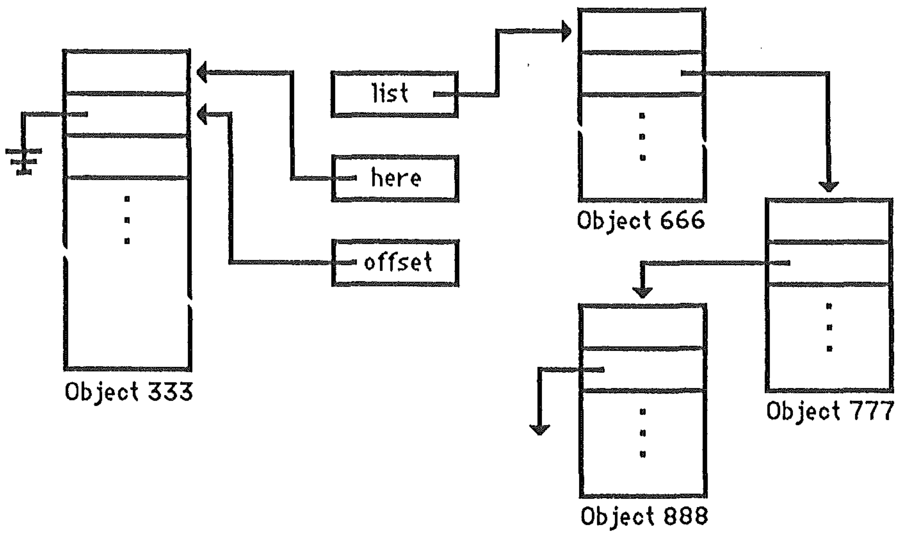

Figure 6.2: State after completing a countDown on object 333 from figure 6.1.

Figure 6.2 above shows the state of the pointers and linked list at the end of performing **countDown** on the object 333 from figure 6.1. The object that is being deallocated by this procedure (here 333) is scanned from its last cell to its first (containing object 666), the cell containing 17 already having had the **countDown** operation performed on it.

At the end of this pass the object 333 will be returned to the head of a free list. As the list labeled "list" in figure 6.2 is not empty the object 666 will be taken as the next to be scanned. Any objects that it points to that have zero count fields, such as 1024, are placed onto the head of the list "list" to be deallocated after the object 666 has been dealt with. This process will continue until all the objects on the list "list" have been deallocated.

Algorithm 6.1 (in Appendix 5) gives the actual Pascal code used in the implementation of this method. It should be noted that the code given in Krasner and McCullough [1985] has been modified slightly as incorrect results resulted from its use as given. The problem was that the statment that reduces the offset (the variable "soFar" in the code given by Krasner) is at the end of the loopin the code given . This leads to class bits being decremented a second time. As, at the end of operating on an object, the class bits no-longer hold an object pointer, incorrect results were obtained. The problem was corrected by moving the decrement statment to directly before the test for completion.

6.4 Pointer Linking.

The third non-recursive method implemented was a direct development from the linear scan of method 6.1.3. It can be seen that the most inefficient phase of this method is the repeated scanning of the ROT to locate objects that have been left in an ambiguous state of allocation by **countDown**, those objects that are neither allocated nor free. To overcome the need to do this scan though the ROT, a linked list structure was introduced to allow the rapid location of secondary objects.

As with method 6.1.3 only one object is deallocated (freed) with any one call to the procedure **countDown**. Secondary objects are kept track of by forming a linked list of *object pointers*. This list was formed in an array of 32k, 16 bit entries which was used for this purpose only. Though more memory is used than with method 6.3, the amount of memory used is known, as there can be only 32K object pointers. As there are are only 15 usable bits in a word, we need only 64K bytes (32K words) to form a complete, explicit stack. Also, with long reference chains less memory may be used for each object being counted down than with the recursive method, as there is no need to push onto the stack the return address, parameters, or any register variables that may be used.

It may at first seem excessive to allocate 32K words to one data structure for the decrementing of reference counts. The explicit stack was formed in this manner for the sake of speed of execution only. Only a very simple conversion from object pointer to "stack" position needs to be made using this technique. Also, if the space had not been allocated to this structure it could only profitably have been used by allocating another segment (number 6) which was usually not required.

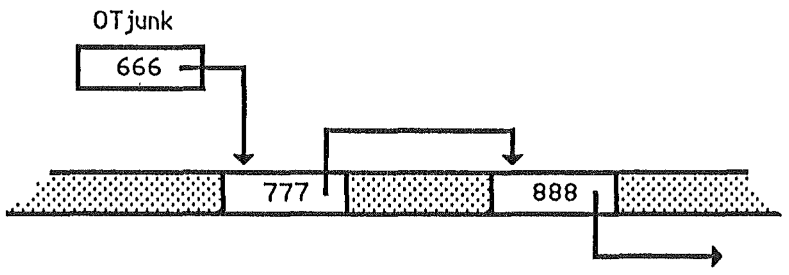

Figure 6.3: The linked list of object pointers for those objects that were secondary to object 333 as it was being counted down.

To deallocate these secondary objects the procedure **seagull** (in a modified form) is used to work though the linked list with OTjunk at its head (figure 6.3), calling **countDown** to deallocate the objects that are on it. Objects that are to be deallocated from the first set of secondary objects (such as object 1024 from object 666) are placed on a second list of secondary objects in the same data structure. This new list of secondary objects is formed using the same global variable (OTjunk) to hold the list head. This is done so that the new list will be deallocated at some future call to the procedure **seagull**.

Shown in figure 6.4 are the two lists of secondary objects, the one that existed before the call to **seagull**, whose current head is now held in a local variable, and the new list being formed as objects from the original list are deallocated. It should be noted that the two lists formed in this way are not connected. If they were it would mean that some error had occurred as an object would have to have been deallocated twice! The Pascal code for these two methods is given in algorithms 6.2 and 6.3 in Appendix 5.

The idea of delaying the freeing of "secondary" objects was first proposed by Weizenbaum [Knuth 1973c], but the lists used contained both forward and backward links (similar to method 6.1.2) and were under programmer control, as were the values of the reference counts. By "under programmer control", it is meant that a programmer using the system (say in Smalltalk-80) could directly affect these two features. This is not the case in the Smalltalk-80 system.

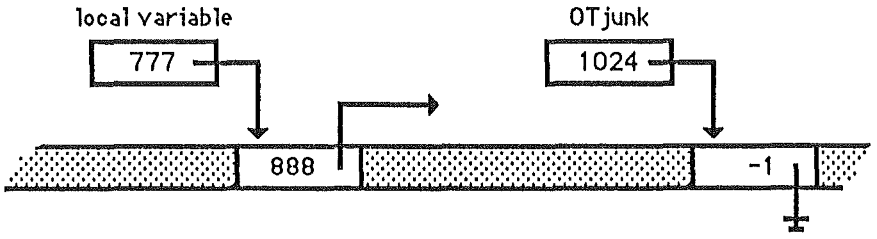

Figure 6.4: The state of the TWO linked lists of objects pointers while the object 666 is being deallocated.

6.5 Performance Comparisons.

The majority of the early work with the system was done using the linear scan method of count reduction (6.1.3). With the move from the virtual memory system to one using only RAM, a change to the pointer-list method (6.4) was made. This change resulted in an increase in execution speed of 8.7%, with both methods being implemented in a combination of Pascal and low-level assembler routines used to access object table data and objects.

A more complete set of data was collected after all changes to the organisation of memory had been completed. Three of the methods described in the previous sections were implemented and tested. These methods were the recursive method (6.1.1) and the two non-recursive methods given in sections 6.3 and 6.4. These different techniques were compared over a number of runs, the results being given in tables 6.1 and 6.2 below.

Run	Method 6.1.1 (recursive)	Method 6.3 (object links)	Method 6.4 (pointer links)
1	531	553	533
2	718	745	720
3	574	588	574
4	319	329	319
5	313	322	316

Table 6.1: Results from the comparison of some methods of count reduction implemented in Pascal. Figures given are the time in seconds for each run.

The three columns of Table 6.1 give the results for Pascal implementations (with calls to assembler routines to handle low level table and object access) of methods 6.1.1 , 6.3 and method 6.4. From the results it can be seen that the recursive method is faster than the object linking of method 6.3. The performance of the pointer linking method 6.4 is however almost the same as 6.1.

Run	Method 6.3 Object Links	Method 6.3 Object Links # 2	Method 6.4 Pointer Links	Method 6.4 with new storePointer
1	499	498	447	401
2	684	684	620	558
3	564	562	522	475
4	302	302	273	246
5	300	297	271	243

Table 6.2: Results from the comparison of some methods of count reduction with implementation in assembler. Figures given are the time in seconds for each run.

Table 6.2 gives the results for methods 6.3 and 6.4 coded in a more efficient manner. The first two columns of the table are for two slightly different versions of method 6.3. The results in the first column come from linking objects in the manner given by Krasner and McCullough, the results in the second column are from linking objects using the special array set aside for method 6.4. The results in these two columns clearly show that the difference in execution speed between methods 6.3 and 6.4 is not due to the differences in linking mechanisms.

The last two columns of table 6.2 are the results gained from the pointer linking of method 6.4. In both these columns the procedure **countDown** has been coded completely in assembler but the **seagull** half of the operation has been left in Pascal. The first of these two columns uses the procedure **storePointer** (see appendix 5) coded in the same manner as columns one and two, that is it follows the code given in Goldberg and Robson and is coded in Pascal. In the last column **storePointer** has been extensively modified to do the minimum amount of work required to perform its function and has been rewritten in assembler.

The major change to the logic of the **storePointer** procedure is that the call to **countDown** has been removed. What occurs instead is that the object pointer that is being replaced has its count reduced in-line. If the count is found to be zero, then the object is entered on the queue with the other objects waiting to be deallocated. The code for this part of the procedure has also been placed in-line.

It can be seen that the assembler versions of these two methods are faster than the Pascal versions; in the case of object linking there is an improvement of around 4.5 percent and for pointer linking of about 9%. This improvement is a result of two main factors, these are:

- the removal of the overhead of calling the assembler routines to do the actual linking etc. Each call removed in this way saves a minimum of 2 instructions (50 cycles if no parameters are passed).
-being able to use registers to hold all the local variables, and so only having to calculate addresses, offsets, etc. once at the start of the loop.

The reasons for the further 9 percent improvement of method 6.4 shown in the last column are the same. The overheads associated with the multiple procedure calls in **storePointer** have been removed.

Some explanation of the difference in speed of methods 6.3 and 6.4 needs to be made. As was shown in table 6.2, the difference in the times measured is not a result of the differences in the linking mechanism. From an examination of the code of the two methods it appears that the rate of execution can be attributed to the relative complexity of the two sections of code. The differences in complexity arise from two of the features of method 6.3:

- from the need in method 6.3 to deal with the class fields of objects in a special, manner and
- from method 6.3 dealing with all objects to be deallocated at any one point in time in the same manner as method 6.1.1. That is, all objects are deallocated at a single call to the procedure.

Attempts to further reduce the amount of overhead involved in calling **seagull** and **countDown** and hence reduce the amount of time taken in count reduction gave no further significant decreases in the time taken to run the Smalltalk-80 system.

Further decreases in the time spent reference counting have been reported [Ungar and Patterson 1982] by changing sections of the interpreter code. How these changes affect performance has not been considered in this work, as it does not strictly come within the framework of memory management.

6.6 Compaction.

Compaction is the process where a large number of free areas of memory are grouped together to form a larger area of free space that hopefully has more utility.

The compaction implementation described in the Blue Book [Goldberg and Robson 1982d] uses three passes of each *segment* to do a compaction. The passes are to:

- scan the ROT to locate all allocated objects in the segment being compacted and set up pointers from these objects back to their fields in the ROT (using the size field) at the same time marking free objects as such.
- move the allocated objects from their current positions in a segment to the lowest position that they can occupy.
- adjust the ROT and size fields so that they again contain the correct information.

This method could be adapted to the linear address space developed in chapter 4, the main obstacle being that either the whole of memory must be compacted or a section must be selected for compaction by testing on the location bits, which would require two comparisons instead of one for every allocated object.

Because a change to a linear address scheme was made in the actual implementation, no memory compaction was used. In addition it appears that the system can perform for quite long periods (in excess of 10 million bytecodes) without any memory compaction being required. In all the runs made with the system no runs, except factorial calculations of values above 250, failed because of a lack of *usable* memory.

Other implementors have also come to the conclusion that a compaction phase is not required, most notable of these being the Berkeley implementation [Ungar and Patterson 1982] which has a memory system almost identical to that developed in section 5.8. There is however no guarantee that this is in fact the case, for example if there is a large degree of fragmentation and a single large object is required.

As an alternative to a full compaction phase the memory states at the end of simulations using the two largest trace files were examined to determine how often free objects were found next to each other in memory. The aim of this exercise was to see if there was any advantage to be gained from building free objects that were neighbours into single larger objects, that is, to find a faster method of doing compaction that was still of some use.

The results obtained from examining the final memory states show quite clearly that there could be some advantage in having this simple form of compaction implemented. Table 6.3 below gives a summary of the results form the two simulations.

	Mscript (section 5.3)	factM (section 5.4)
number of runs	24	20
number of runs of only 2 objects	16	10
total number of objects in runs	72	246
total amount of space in runs	6270	4923
average size of run (> 2)	433	459
total number of free objects in system	165	284
total amount of free space in system	39082	39831

Table 6.3: Summary of data on the runs of objects found in simulation runs of Mscript and factM.

The rows in table 6.3 represent the following,

number of runs	the number of times that free objects were found next to each other in the memory dumps that were done at the end of the simulations. Free objects that are next to each other form a "run" of objects.

runs of only 2	the number ofruns that consisted of only two objects.
total number of objects	the total number of objects that were part of any run in the simulation data.
total amount of space	that sum of the sizes of the objects that were contained in runs.
average size of run	the mean size, in words, of the runs that contained more than two objects.
free objects in system	the total number of all free objects, of all sizes, in the system.
free space in system	the sum of the sizes (in words) of all free objects in the system.

Table 6.3 shows there are a large number of free objects that are found next to each other. What demonstrates the usefulness of this method of compaction is an examination of some of the individual runs that occur.

From the Mscript simulation four runs of objects are of particular significance, these runs could be used to form large objects in the following manner,

$$114 = (4 * 8) + 16 + 66$$

$$180 = (20 * 9)$$

$$663 = 218 + 264 + 181$$

$$2608 = 396 + 2212$$

For example, the data presented on the first line mean that there were 8 objects of a size of 4 words followed by two objects of 16 and 66 words, which can be combined to form a single object of 114 words.

The last two examples of the runs shown from Mscript shows that some quite large objects can be formed by joining a number of objects which are already of a reasonable size.

The data obtained from the calculation of factorial 200 is more convincing about the usefulness of this method. The following runs represent the largest objects that can be formed from free objects in this run.

107	=	(20 * 5) + 7
2947	=	27 + (20 * 146)
997	=	(30 * 30) + 5 + 6 + 6 + 7 + 8 + 8 + (9 * 3) + (10 * 3)
179	=	35 + 35 + 36 + 36 + 37

From the data given, the manner in which factorials are calculated becomes evident. There are a large number of context objects of size 20 (181) present in the runs, indicating the recursive manner of calculation. More important is the manner in which space is allocated for the large positive integers used to hold the result of the calculation. It can be seen that as the calculation has progressed, larger objects (by 1 word) have been allocated to contain the increasing number of significant digits of the result. When the result outgrows the space that an object provides that object is then discarded. This feature of massive fragmentation of memory is supported by evidence from actual failures of the Smalltalk-80 system due to a lack of *suitable* memory being available when calculating the results of factorial 500 and factorial 1000. In the dump of the list of large free objects that these two runs produced, the series nature of allocated objects is obvious. The head of the free list starts with objects of just over 160 words and follows a strictly deceasing linear sequence, as in the fourth example of runs from factM, down to a value of 41 at the tail, this being the minimum value that objects in the list may have.

6.7 Observed Count Reduction Behaviour.

It was observed that a large percentage of the values being replaced by calls to **storePointer** have count values that have reached the overflow point (count = 255). The majority of these objects have the object pointer value of 1and the object that is associated with this pointer value is the Null object. This seems to show that a large proportion of storePointer activity is used in replacing the values that are initially placed in objects at allocation.

An examination of the Xerox Virtual Image shows that of 77787 words contained in all pointer objects, 12.5% have either a small integer value or the value of the Null pointer. By placing hooks into the **storePointer** procedure it was further found that 82% of all calls were to replace objects that had pointer values less than 27 (objects 0 to 26 have count values of 255). Also, 79% were calls to replace objects with pointer values of 1 or less, and 48% were calls replacing the Null value pointer. The code as implemented in **storePointer** currently makes use of this information by not trying to decrement the count field of objects with pointer values less than 27, but no measurable inprovement in execution speed was achieved by this modification. This is probably due to having to use a **CMP** instruction in place of the faster **TST** (for zero) inthe assembler code.

CHAPTER VII

MISCELLANEOUS RESULTS

7.0 Introduction.

During the implementation of this Smalltalk-80 system several areas other than those discussed in the previous chapters were examined. These areas are covered briefly in this chapter because they:

- have a small effect on the overall performance of the system,
- they are simply stated and do not warrant separate chapters, or
- they are of a speculative nature.

7.1 Flag Tables.

The initial decisions regarding the structure of the flag arrays were discussed in section 3.4. By using the debugger provided with the system it was determined that access to these tables (packed arrays of boolean)provided by the Pascal compiler was by means of **ROR** instructions (rotate right), used to bring the required bit into the zero position of each byte so that it could be tested. Associated with this operation there were overheads in calculating the address of the byte holding the flag and the bit placement in the byte of the 16 bit integer used to access the array.

To try to avoid these overheads, two other structures for the flag tables were examined. Both of these structures used byte arrays allocated on the heap and accessed via calls to assembler routines. It was hoped that the added time introduced by the procedure calls would be more than offset by the elimination of the complex mechanism used to perform access to the packed arrays of boolean.

The two alternative structures were:

- allocation of one complete byte array for each of the flag bits, ie. the F, 0 and P bits, the flag being set when the value of the cell was non-zero.

- the use of a single byte array to contain all three flags, using one bit in each byte for each flag value to save space (64K bytes total).

Results from timed runs using these three methods showed that there was no detectable difference in the speed of the implementations.

One further change was made to the system of flags used. Because of the nature of the count reduction mechanism used (section 6.4) procedures that used the value of the Fbit to determine the existence of objects such as firstInstOf, were required to not only test the Fbit to determine if an object was allocated. but to test the object's count bits to check if the object had a zero count but was not yet deallocated. Due to of this extra test the Fbit became redundant. The removal of all code related to this one flag resulted in a small (3-4 seconds of the total run time, around 1%) decrease in the time spent executing the set of standard benchmarks used. It should be noted that the test on the count bits is slightly faster than a test on the flag bits in the second method implemented. This is because a **TST** instruction is used to compare the count against a value of zero and the flag bit is tested with an **AND** instruction which involves the use of an intermediate value.

7.2 The Effect of Procedure Calls.

As was shown in section 6.5, the removal of procedure calls can have a significant effect on the operating speed of the system. These results are confirmed in part by the conversion to assembler of several other small memory manager functions. Calls to these new assembler routines were then made directly from the interpreter code. Table 7.1 shows the results of some of these changes made to the system.

Procedure Name.	Number of Places Called from Interpretor code.	Percentage Performance Gain.
storePointer	53	4.5
fetchClass	32	1
IntegerObjectOf & IntegerValue of	15	2.5

Table 7.1: The effect of conversion to assembler of some sections of code.

Improvement was also gained from altering the interpreter code written by Mr. P.D. Anderson. By converting all single line procedures and functions (those which have their entire body in one Pascal statment) to inline code, a performance gain of 17% was obtained. Further gains may be possible in this area by converting calls to assembler subroutines to inline code, removing completely the overhead of procedure calls. The prime example of a function where large gains can be made is the function **fetchPointer** which accounts for the largest percentage of the calls made to interface procedures from the interpreter. This step was not taken, as a large degree of readability of the code would have been lost, along with some of the ability to place hooks for measurement in the low level code of the system.

As an example of how the overhead of procedure calls can affect the speed of the system, the section of code that draws windows onto the screen may be considered. Each window is a white area surrounded by a black border. To draw such a window the system first draws a black rectangle on the screen in the position that the window will occupy. Inside of this black area a second, slightly smaller white area is drawn. The net result of this procedure is a white window with a black border.

When the system was first implemented, this window drawing was done via calls to the **storeWord** procedure. This procedure places the required data into one 16 bit word of memory with each invocation. Because of the repeated calls to this procedure, the display of each window on the screen occupied several seconds for each of the phases involved. As a solution to the performance problem presented by this situation, the function **address** was introduced so that the interpreter could request the location of an object and then operate on it directly without any further action on the part of the memory management code. The result of this change was to allow windows to appear at the maximum speed possible. For large windows each phase of the window drawing procedure is still distinguishable, but the overall time taken is only about one second. For small windows (such as the header tags for main windows) the construction of a window appears instantaneous and the separate phases are not distinguishable.

7.3 Locality of Execution.

Because of the low execution speed of the virtual memory system implemented, as detailed in chapter 4, and because of the general belief that the locality of execution in the Smalltalk-80 system is much lower than in more "traditional" systems implemented in languages such as Fortran or Pascal [Ballard and Shirron 1982], a brief investigation of the object reference behaviour of the system was made.

The actual measurement being made is of the working set size of the system as described by Denning [1968]. The concept of a working set has been used in such areas as the measurement of data reference behaviour in data base systems [Rodriguez-Rossell, Kearns and DeFazio 1983] and there appears to be no obvious reason why it cannot also be applied here.

Denning defines the working set size w(t,T) at time t as

w(t,T) = number of pages in W(t,T)

where

T = the size of the window looking backwards from time t.

W(t,T) = the set of information accessed over the period T.

Both T and t have been defined here in terms of byte codes executed, and the number of 16 bit words accessed (the sum of the sizes of each of the objects accessed) has been used in place of the number of pages accessed. The number of objects accessed was not used as a direct replacement for the number of pages as objects, unlike pages, are of variable sizes.

The window sizes were chosen to represent the time interval of approximately 1 second for a number of different systems. The systems that were considered when choosing a range of values are listed in table 7.2,

system	maximum reported byte codes / second.	reference.
Apple Lisa	1,100	this work
68000	3,000	McCullough 1983
PDP 11/23	5,000	Ballard 1983
VAX 11/780	15 to 25,000	Ballard 1983
680x0	60,000	Wirfs-Brock 1984

Table 7.2: Reported performance results used to determine window sizes. The exact chip used in Wirfs-Brock84 is not known. It may be any of 68000, 68010 or 68020, but because of the high speed of the system the latter is felt to be the most probable.

This gave the set of window sizes of 1, 5, 10, 20 and 50 thousand byte codes. A set of three runs for each window size were made over a total of 35 million byte codes. The data from these runs was used to produce figure 7.1.

The manner in which measurements were made was to place "hooks" in the most frequently used areas of code that accessed individual objects. To locate the sections of low level code that were most often used, a series of runs were made over a total of 2.8 million byte codes with the system performing functions such as calculations (1+2+3, 200 factorial etc) and decompiling sections of compiled Smalltalk-80 code.

To gather this first set of data, a short section of code was added to each of the low level assembler routines. This section of code incremented a counter associated with each procedure that was located in global memory . The results of this first part of the investigation are shown in Table 7.3 below.

Procedure or function name.	Total number of calls.	Percentage of total calls.	Hooks for second part were placed in these routines.
address	13,295	---	*
byteGet	3,044,688	10.4	*
bytePut	17,468	---	
ftchWord	135,500	---	
ftchPtr	11,343,790	38.8	*
getFloat	2,649	---	
putFloat	1,326	---	
wordGet	9,570,096	32.8	*
wordPut	5,082,434	17.4	*
	---------	----	
	29,211,246	99.4	

Table 7.3: Frequency of use for the lowest level interface routines.

It can be seen from the data that four functions account for almost all accesses to objects (99.4%). The call to the function **address** was included in the set of routines to monitor because this is the main mechanism used to access the screen object. As this object is one of the largest in the system it was felt that it should always be included when determining the working set.

The addition of the locality measurement hooks reduced the maximum bytecode execution rate from a maximum 1100 bytecodes per second to 430 bytecodes per second. This came about because the hooks added to the procedures accounted for 50% of the code in those procedures.

The technique used to measure the reference behaviour of objects in the system was to use a 32K, 16 bit array to hold a copy of the size bits of each object accessed over a "time" period of N byte codes. That is, on each access of an object the value of its size field was stored in the cell of the array that mapped onto its object pointer. At the end of each time period, the sum of the object sizes is computed and the result stored. During this summing process each cell of the array is set to zero, so that at the start of the next time period the working set size will be zero. The average values of these measurements were used to produce the curves shown in Figure 7.1.

The three curves drawn on the graph represent the three types of runs that were made to measure locality. These runs can be broken down into two classes, those that were numerically intensive and one run that was interactively intensive. In the first class were the following,

- "Pen new mandala: 20 diameter: 250". This draws a mandala object on the screen and access to the screen object is frequent unlike the second run in this class,
- "250 factorial". This has minimal interaction with the screen but uses a large number of objects to perform the calculation.

Both of the runs above were made over a total of 3 million bytecodes. The second class of run was made using the system browser to Smalltalk-80 methods (the same with each run), decompile the methods to Smalltalk-80 text, modify the text and then re-compile it. This set of runs thus involved three large Smalltalk-80 programs and measured their locality, the programs being:

- the system browser,
- the de-compiler and
- the compiler.

All three of these programs, apart from being the largest readily available, are central to programming in the Smalltalk-80 language. The items of code that were chosen to be modified and re-compiled were also part of the code for the compiler.

A larger set of data than that gathered would have been preferable but the time taken for each of the runs involved proved to be excessive. The runs using the system browser, with a window size of 50,000 bytecodes, runs took around 1 hour 40 minutes to complete. Using the the smaller window sizes some runs were over 3 hours in length.

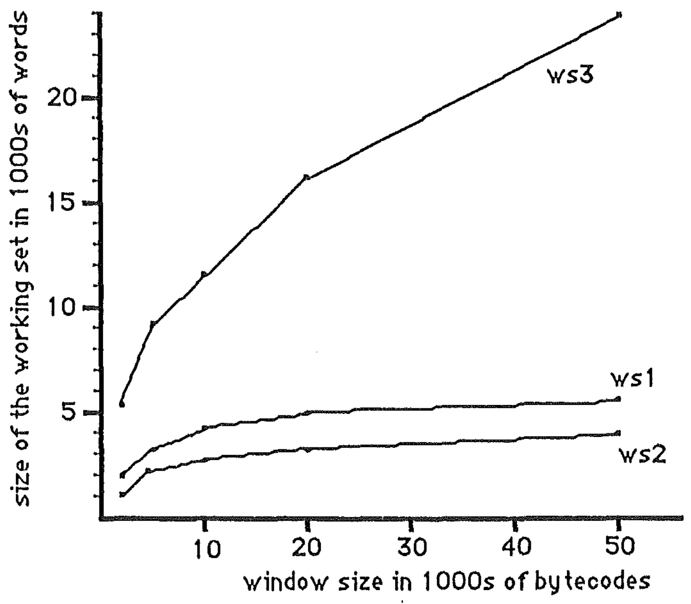

Figure 7.1: Curves produced by measuring the working set of a running Smalltalk-80 system. The curve WS1 is produced by drawing a mandala, WS2 was produced by executing "250 factorial" and WS3 comes from the execution of the system compiler/decompiler.

Figure 7.1 shows the same behaviour as the one given in Denning for the predicted behaviour of w(t,T). This is shown most strongly in the curves for the two numerically intensive sets of runs. The curve for the interactive set of runs shows only the start of the curve expected. For a program that showed no locality of execution a straight line would be expected for the plots. None of the curves shown here, however, can be considered to be a linear function of the window size. Rodriguez-Rosell in his discussion of locality in data base system gives an example of this type of linear plot. In the example he uses there is a definite linear relationship between the working set size and the window size. As this does not appear to be the case in the Smalltalk-80 system, for mthe measurements made, it can be assumed that for a sufficently powerful, system the application of virtual memory techniques would be feasible.

However a warning with regard to this conclusion must be made. The set of programs over which the measurements of locality were made do not include examples of programs such as simulations or large list processing applications. It is possible that programs of this type would have locality profiles that would not follow the curves given in figure 6.1. Examples of this kind of application have not been included because of the low execution speed of the system (table 6.3). Ballard describes the execution speed of their PDP11 version of Smalltalk-80 by commenting that it "...resembled molasses in December". Even dedicated Smalltalk-80 workstations such as the Xerox Dolphin have been described as being slow [Wyvill 1986]

7.4 Parallelism.

It has been seen that garbage collection in Smalltalk-80 is very important. Because of this the possibility of providing more power from a system by having more processing sites was examined. In the words of C. Gordon Bell "The number of processors potentially has the greatest effect, because it can be quite possibly be increased indefinitely" [Bell 1986]. While the level of parallelism in the Smalltalk-80 virtual machine is small compared with large scientific problems there does exist some possibility for exploiting it.

In what follows, it is assumed that the hardware for the system will be specially built (or modified) for this system and that the Smalltalk-80 virtual machine will be the lowest level of code in the system.

The Smalltalk-80 system has been designed in two separate parts, the interpreter and the memory manager that are joined by a set of interface procedures that control their interaction. There are two aspects to this rigid method of communication between the two sections of the system. On the positive side they allow for ease of implementation by forcing a modular approach to system development. The negative side to the interface is that it results in a large number of procedure calls that in many cases could otherwise be avoided.

The two areas of the memory manager's functions that can gain most from this approach are

- maintenance of system memory and
- object deallocation, because of the large amount of work that must be done that is hidden from the interpreter.

Maintenance of system memory involves functions such as the sorting of free lists (section 5.10) and the running compactions examined in Section 6.6.

Of these two areas, the most gain will come performing the deallocation of objects in parallel with the functions of the interpreter. This is because of the pointer chains and multiple objects that must be accessed. What follows is a discussion of possible methods of introducing parallelism into the Smalltalk-80 system and their effects.

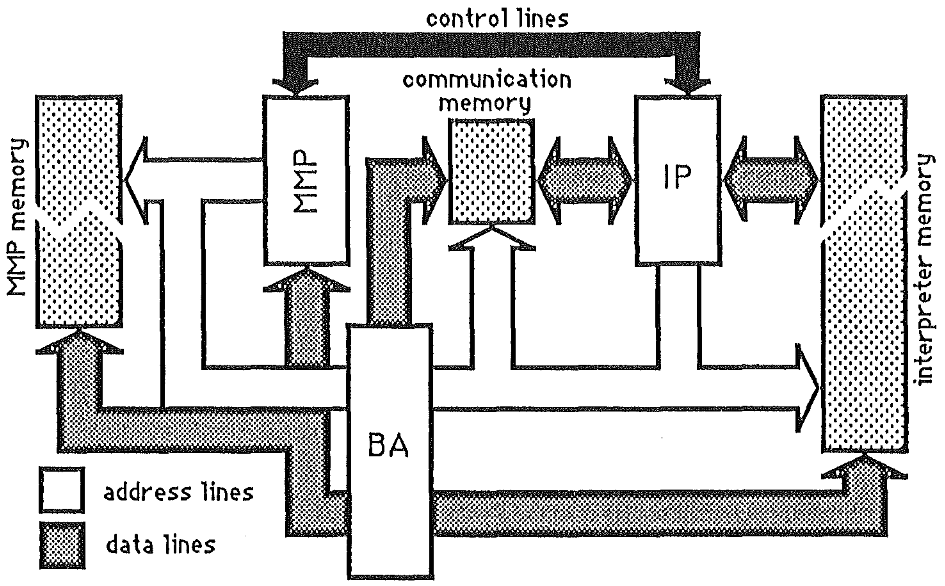

Figure 7.2: An idealised diagram of a two processor system for Smalltalk-80. The block labeled "BA" is for Bus arbitration.

The system considered has a minimum of two processors (see figure 7.2 above). One processor is used by the memory management code that is directly involved in the maintenance of the system's Smalltalk-80 memory space. This processor will be termed the MMP. The second processor will perform all the functions of the interpreter. The scope of the interpreter's functional domain will be expanded to include all routine access to non-header data of objects. This processor will be termed the IP.

Associated with each of these processors with be at least one area of memory. The MMP will have associated with it a small area of private memory (0.5 megabytes) that will be used to contain

- its code
- the system tables, such as the count bits, that must be kept hidden from the IP.
- the lists of linked objects similar to the table structures introduced in section 6.4 for linking objects to be counted down.

A variant of this table could be used to form linked lists of free objects rather than use the objects themselves (figure 7.3). This would help reduce the number of accesses that the

MMP would have to make to the Smalltalk-80 address space, which is controlled primarily by the IP.

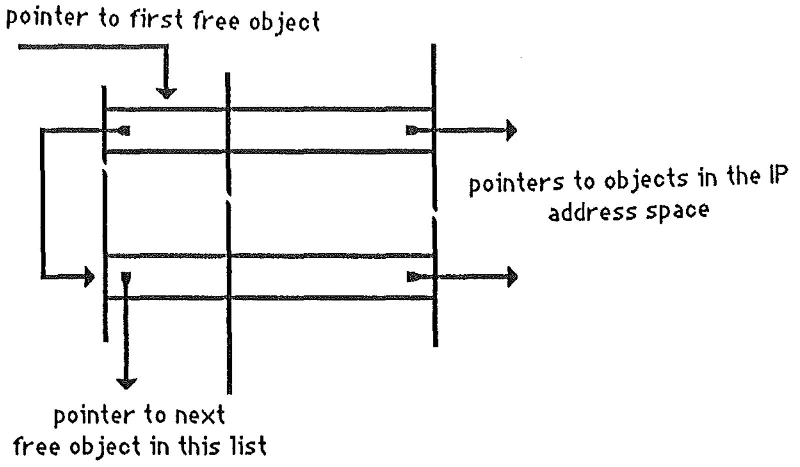
pointer to first free object
pointers to objects in the IP address space
pointer to next free object in this list

Figure 7.3: A possible method of reducing MMP accesses to the IP address space by forming links of free objects in the MMP memory space.

The IP would have a much larger area of memory associated with it. This area would contain all the space used for objects in the Smalltalk-80 system, as well as the code used for the interpreter and any tables, such as a copy of the location bits, that it required. This area of memory would also need to be accessible to the MMP as data in the two heading words of objects in the system would have to be periodically updated. It may even be possible to move these two header words, or copies of them, to the memory manager's address space.

The functions that the MMP would perform are those of maintaining the memory space of the system and data such as the count bits of each object. The location bits table, as it needs to be accessed by both processors and frequently by the IP, would have to be duplicated in each of the processor memory spaces. Having duplicates of this table would allow the IP to access objects in a routine manner without having to gain access to information from the MMP. This duplication represents no problem in the operation of the system as the location information is relatively static and only changes when,

- objects are split, this requires an addition of one entry in the table, and modification of the two objects affected.
- when objects are merged, this requires that the header data of the lead object is altered.

- when full compaction is done, in which case all operation of the IP will be halted.

The first of these situations is a result of a request for space by the IP and the IP will be waiting on action from the MMP so the necessary changes to tables and objects in the IP data space can be made before IP execution resumes. The second case involves only free objects that the IP will not be accessing. If an access from the IP does occur on these objects then an error has occurred as the count of the objects has been reduced to zero while a pointer to the object still exists and this situation should never occur. To perform the change the MMP will either have to wait until the IP asks it to perform some other function or the MMP will have to interrupt the IP itself.

The third case is rather more involved as it results in large scale changes to the structure of memory. The complication that could arise here is that the IP code could have the *address* of some object stored in a local variable. Compaction of memory would require this variable to be updated in some manner, unless the interpreter code was structured in such a way that compaction could not occur when addresses were cached in this manner.

There only two reasons for the IP to communicate with the MMP, they are to either allocate an object or to change the reference count of an object. The process of allocation has a simple flow of control,

- IP asks for an object of some size X.
- MMP searches for an object to fill this request.
- MMP adjusts whatever values it needs to in the IP address space (location bits).
- MMP passes the value of the object pointer to the IP which resumes execution.
- MMP fixes up any tables, lists etc. in its own address space.
- MMP resumes what it was doing prior to the request.

The second reason for interrupting the MMP is to change the counts of objects either directly via calls from the **decRefsTo** or **countUp** procedures, or indirectly via a **storePointer** function being executed. In the first two cases the IP would in some manner "interrupt" or signal the MMP that this was to happen and pass it an object pointer value. The IP will then resume its processing and the MMP will deal with the matter from that point. The case involving **storePointer** is slightly more complex but not greatly so.

The process of storing a pointer could be broken into two parts. The first of these would be performed by the IP and would be to retrieve the value that was being replaced and hold it in local storage until it was passed to the MMP. It would then place the new value of the cell into that cell.

The second part of the operation would come when the IP interrupted the MMP and passes *both* values to the MMP process and continued with its own execution. The MMP, when it received these values would then proceed with the remainder of the **storePointer** operation.

In the implemented assembler version of storePointer there are a total of 41 instructions used to do the actual processing. Of these 8 are used to fetch and replace the correct values and cannot be performed in parallel with further IP execution. In addition there are a further 11 instructions used for procedure entry and exit and saving of register values. The instructions used for procedure entry and exit can be disregarded if we assume that the code for the IP section of storePointer is to be done inline. If this is actually the case then the instructions for saving registers can possibly be disposed of as well (this depends on how the IP is coded; it is assumed that assembler would be used because we are building special hardware and the extra effort would be small).

The process of counting down reference counts of objects is more complex. Because of the links between objects in the Smalltalk-80 system, the MMP has to access the body of objects that are being deallocated so their pointer chains can be followed to secondary objects. The result of this is that the MMP will have to interupt the IP to perform these accesses. The effects in terms of non-overlapping code should, however, be small, limited to making a copy of each object that must be accessed which can be done with a tight loop. On examination, the shortest path that can be taken through the code of **countDown** (algorithm 6.4, appendix 5) by a non-pointer, non-method object that's count bits are equal to 255, takes 45 instructions (not including procedure entry or exit). Of these, only two instructions examine data in the object. If the object being examined is a pointer object of three words, then around 85 instructions must be executed (depending on the status of the free list) of which at least three must access the object itself. This depends on how free objects are linked; if they are linked via their class fields then this figure rises to nine. One further point must be taken into consideration - in Section 6.7 it was noted that a large amount **storePointer** activity was due to replacing values that could not have their count fields decremented. The direct effect of this is that fewer accesses to memory would be needed.

If the MMP process has sufficient time, it can undertake to do extra work that could not otherwise be done in a Smalltalk-80 system. In Section 5.10 it was found that a sorted list of large objects resulted in a more efficient use of the memory available to the system. Also, in Section 6.6 it is shown that some large blocks of space could be reclaimed by the joining of free objects that were next to each other. Both these functions could be performed in the "background" if a multiple processor system of the type described was used.

Because allocation and **storePointer** events are frequent (2 **storePointer** calls per bytecode on average) some form of synchronisation would be required to ensure that the MMP always does some minimum amount of work before being interrupted for the next transaction. On a system based on MC68000 chips this could be achieved by having the MMP operate in the supervisor processor state when dealing with with any operation that should not be interupted. At the end of this critical code, the processor state would change and the MMP would continue with some task that did not require that it was not interupted. The IP could detect operation of the MMP in the supervisor state via a set of three pins that are set to reflect the state that the MMP is operating in. The IP could examine the value of the pins via a two instruction loop before interrupting. This looping of the IP should not affect the operation of the system as a whole.

The means of communicating would not need to be via interrupts. Shared memory that could be read at the same time by both processors could be used as a set of mail boxes, dual port memory chips that use duplicate sets of address and data lines of the type required are available [Page 1986]. When a mail box was empty it would contain negative values indicating the absence of object pointer values. The MMP would execute in a busy-wait loop, checking for incoming mail in boxs at regular intervals. The IP would now be under a less strict set of conditions. To change count values it would need only to post an object pointer value in a mail box. If no mail boxes were available it would then wait. Possible problems with objects being incorrectly deallocated could be avoided by performing all **countUp** operations before **countDown** operations.

A third, and possibly more attractive option for hardware in a multi-processor system would be to use Transputer devices (Inmos Inc.). As these devices have build in bi-directional data paths and have been specificly designed for inter-processor communication, in that they have machine level primatives to handle synchronisation, they seem ideal for the task of distributing the computational workload.

It may also be feasible to build a system that effectively utilises more than the two processors that have been used above. Some of the areas where it may be possible to introduce further processors are listed below,

- a processor could be used to handle the graphics display for the system.
- a processor could be placed between the MMP and the IP to act as a secretary for the MMP this would be to ensure that the IP never has to wait to deliver instructions and data to the MMP.
- a processor could be used to keep trace of all the input comming from the user of the system (keyboard and mouse). At present the interpreter checks that there has been no input via a loop.

Chapter VIII

Summary and Conclusions

8.0 Summary.

The work that is presented here went through three main phases. The first of these, from March 1985 to June 1985 consisted of understanding the system and what it did. As there were two people working on different parts of the system the learning process was somewhat slower than it should have been. This phase started with an examination of the Berkeley BS II Smalltalk-80 system for the SUN II workstation, the intenshion being to port this system to the Lisa. This approach was abandoned for two main reasons. The first of these was that the BS II system is written in C and is designed to run on top of the Unix operating system. The first problem would have been to replace all Unix system calls in the code. For example the BS II code performs allocation via the malloc system call [Ungar and Patterson 1983], this would have to be simulated in Pascal. To do this effectively it was thought that most of the features of the low level Smalltalk-80 memory management code would have to be implemented. For example, if an object of 64K words were requested, on the SUN system it requires one malloc call. On the Lisa it would require four calls because of the 32K byte limit on the NEW statment.

The second problem was that the only high level language available for the Lisa was Pascal, a C compiler for the Lisa that we had hoped to obtain being canceled with the introduction of the Apple Macintosh. An attempt was made, however, to build a program to convert C to Pascal [Carnevale 1985]. The major problem with the C code was the amount of it that was contained in macros. A macro pre-processor could have been built but would have taken more time and effort than was thought acceptable. Because of these problems the BS II implementation was abandoned in favour of an implementation straight from Goldberg and Robson.

The second phase involved the implementation of the Virtual Memory system detailed in chapter 4. The work with a working system spanned the time from August 1985 to 20th of December of the same year. Work on the segment paging system (section 4.15) was started in late October and continued through to the introduction of a completely RAM based system in late December. At the same time as work was being done on the segment paging system the first set of simulations were being carried out with the trace file "s100".

The third phase involved the implementation of the RAM only system. The initial work on this was completed over Christmas 1986 and New Year 1987. The RAM only system was made possible by receiving a memory board from the U.S.A. that was used to replace one of the Apple memory boards. Initial performance from the RAM system was disappointing. Benchmarks showed that it was only twice as fast as the virtual memory implementation (about 270 bytecodes per second), but loading of the Virtual Image was much faster at 196 seconds.

The second set of simulations were conducted at this point, as disk space was no longer needed to act as memory for the Smalltalk-80 system. The much longer trace file "Mscript" was taken at this time. Further advances in the work that was undertaken from January 1986 to September 1986 came from three main areas. They were;

- the introduction of multiple free lists as described in section 5.8.
- the introduction of the pointer linking method of garbage collection that was introduced in section 6.4.
- the conversion of Pascal code to assembler.

The last of these points is very important, for example the increase in execution speed introduced by converting the code of the pointer linking method from Pascal to assembler was about 26%.

8.1 Conclusions.

Firstly some observations about the Smalltalk-80 system as implemented:

- it creates a fantastic amount of garbage - as noted there are on average two **storePointer** operations for every bytecode that is executed.
- the system needs a lot of processor power as, because it is an interpreted system, there are large overheads associated with each and every Smalltalk-80 operation. The Lisa computer was just too slow to produce an effective implementation.
- it needs a lot of memory - the 5 segments that could be allocated to the system were only just enough to run the system.
- it is a very robust system. One error in the low level memory management code that was made during coding in May/June 1985 did not appear until early in October of that year!

The conclusions reached about the system are that:

- the memory management system proposed by Goldberg and Robson does not go far enough. It was observed during the course of this work that most allocation / deallocation activity that occurred was because of objects 20 words in size. Table 5.6 in section 5.10 shows how badly affected a system that conforms to Goldberg and Robson will be.

- if reference counting is used then the garbage collector is the most important section of code in the system. Its performance (or lack of it) will make or break a system. Duff [1986] states that "...unoptimised reference-counting collectors can consume up to 70 percent of total execution time because of the constant maintenance of reference counts". The Smalltalk-80 Version 1 (this implementation is of Version 2) implemention for the Apple Macintosh Plus uses a marking garbage collector rather than reference counting [APPLE 1986]. The system is much faster to use but it suffers from making the user wait for long periods (10 minutes +) when it runs out of usable memory. Multiple processors have been examined in Section 7.4 as an alternative.

- implementing virtual memory with software is not worth the trouble unless there is no alternative. Similar problems with performance seem to have been encountered by Dijkstra when implementing the T.H.E. system [Parnas 1986]. It is possible that many of the problems that were encountered were due to the slow speed of the hard disk ("a disk for storing things on, not a disk for paging" [MacLean 1985]).

8.2 Features not Implemented.

The system as implemented is a full Smalltalk-80 Version 2 implementation and is more complete in some ways than the Smalltalk-80 versions for the standard Macintosh Plus which has some methods and classes removed. However two important features were not implemented.

The first of these was the file system. Because of this the system is of no use as a system for using the Smalltalk-80 language and environment. This was however of little concern. Because of the speed that the system operates at it is doubtful that anyone would want to use it for development work. After the initial conversion to a RAM only base with its slow execution the implementation of the system came to be viewed as a tool for examining the workings of Smalltalk-80 and not as a useful system in its own right.

The second item that was not implemented was the "snapshot". A snapshot is the means by where the whole of the Smalltalk-80 system state can be saved. The aim is that the system, should it go down for any reason, can be brought back to a working state at the point where the last snapshot was taken. At no time during its development was there any means to save the state of the system. For the work presented in this volume that was of no concern as all that was required of the system was that it performed as a Smalltalk-80 system that could be measured, allowing changes in performance due to changes in code and data structures to be observed.

8.3 Final Thoughts.

The "finished" system is still quite slow. Its speed of execution however could still be improved by converting to assembler critical sections of the bytecode interpreter such as the bytecode dispatch routines. Improvement could also come from introducing many of the optimisations that are used in the BS II Smalltalk-80 system.

As a tool for examining in inner workings of the Smalltalk-80 system the system has been adequate. Some of the work can be put into perspective by results from the Tektronix Smalltalk-80 group which consisted of three software people plus other people to help with construction of hardware, CPU, bit mapped displays etc. Below is a table compairing the performance figures reported in McCullough [1982.1] with the final system inplemented at Canterbury.

System.	Speed in Bytecodes per Second.	Processor Type and Clock Rate.	Languages Used.	Notes
C1	900	4.5 MHz 68000	Pascal, assembler	average value
T1	470	8 MHz 68000	Pascal	
T2	800	8 MHz 68000	Pascal, assembler	instruction cache
T3	3500	8 MHz 68000	assembler	Instruction cache

Table 8.1: Comparison of Tektronix and Canterbury 68000 Smalltalk-80 systems. C = Canterbury and T = Tektronix. No details of the instruction cache are given in the paper.

Appendix I
Hardware

The System.

The machine that Smalltalk-80 was to be implemented on was an Apple Computer Lisa II, which is also known as the Macintosh XL. The hardware of the system is comprised of three separate items, they are:

-the system box containing the bit mapped screen, processor and disk drives.

- a 76 key detached keyboard.

- a *one* button mouse.

The main processor of the system is a Motorola MC68000 chip operating at a clock speed of 5 MHz. However because the 68000 is also called upon to perform the function of generating and maintaining the video image the clock speed is effectively reduced to 4.5 MHz.

System memory supplied by Apple is 1 megabyte of RAM (with parity) on two boards using 64K DRAM chips. This was later upgraded to a total of 1.5 megabytes. This was done by replacing one of the Apple memory boards with a Ramstak board from AST Research. This new board was capable of holding a maximum of 2 megabytes using 256K DRAM chips. Only half of this capacity was used however, giving a total of 1.5 megabytes of RAM (1 + 0.5).

The Lisa has a number of other processors associated with it [Morgan 1983]. There is a processor in the keyboard assembly that is used to scan the keyboard for input an there is also a processor used to collect commands from the keyboard and record mouse events. This allows the mouse button to be "shifted" so that by using the "option" and "apple" keys we have the three buttons that a Smalltalk-80 mouse is required to have.

The Lisa has two disk drives. The main drive is a 10 megabyte Winchester hard disk and the other is a 3.5 inch single-sided, double-density floppy drive capable of storing 400K bytes (formated). The floppy disk drive is the same as those used in Apple's Macintosh computers. This allows files to be transfered between these two machines. This is also the mechanism used to transfer the Virtual Image from a VAX 11/750, and the trace files (Mscript etc.) from the Lisa to the same machine using the macget/macput set of programs (4.2 BSD Unix).

The Lisa's screen is a black and white bit-mapped display with a resolution of 720 x 364 pixels with a refresh rate of 60 Hz. The pixels are not square and thus figures such as circles are drawn in a distorted manner by the Smalltalk-80 system.

For communication with other systems the Lisa is equipped with 2 serial ports capable of a maximum of 19.2K baud in either asynchronous, byte-sync or bit-sync protocols. In addition there are three I/O expansion slots directly connected to the system bus capable of DMA transfers.

The 68000.

This chip comes in a 64 pin package, the chip itself is a 32 bit processor (internal) with a 16 bit data bus and a 24 bit address bus and is capable of linearly addressing 16 megabytes. This address space can be expanded to address 64 megabytes in 4 banks by using the three pins used to communicate the processor status of the machine. There are four states that may be used in this way, being the supervisor and user program and data spaces. there are also two "reserved" states for the user and one reserved and one interrupt state for the supervisor [Jaulent 1985].

The 68000 chip is normally used with a 8 MHz clock, as opposed to the 5MHz clock used on the Lisa. This gives an execution time of 0.5 microseconds for the execution of the fastest instruction [King and Knight 1983].

The machine architecture is two address, either register-memory, memory-register or register-register. The 68000 has 18 registers:

- 8 data registers, 32 bits wide named D0 to D7.
- 9 address registers, 32 bits wide named A0 to A7, the register A7 being duplicated for the user and supervisor states.
- 1 status register, 16 bits wide, used to hold system flags.

Instructions can operate on 8, 16 or 32 bit quantities. The exceptions are that there are no 32 bit multiply or divide instructions and byte operations are forbidden on address registers, as are some other instructions.

The feature of the 68000 that had the most effect on the implemented system was the manner in which offsets in code and data were made. In both cases offsets are 32K bytes in either a positive or negative direction. Because of this the Pascal compiler in use divided code and data into 32K byte segments. The results of this are that:

- global memory was limited to 32K bytes maximum. This forced parts of the ROT onto the heap which made access more complex.
- all offsets, including those to 64K byte arrays such as the list of free pointers (important for the work described in sections 5.4 and 5.5) had to be made using long word offsets. This slows the system as long word offsets have to be constructed in two or more steps. For example, CLR the offset register, MOVE the 16 bit offset into it, use result as a long integer.

There are several members of the 68000 family; their relative performances are compared in table a1.1 below which is adapted from MacGregor [1985].

clock speed in MHz	68000	68010	68020 0% hit	68020 64% hit	68020 100% hit
4.5	0.36	---	---	---	---
8	0.64	0.66	---	---	---
10	0.80	0.83	---	---	---
12.5	---	1.03	1.63	1.74	1.96
16.6	---	---	2.16	2.33	7.62
mean clocks per instruction	12.576	12.107	11.676	9.975	7.424

Table a1.1: Relative performance of members of the MC680x0 family of proccessors, values in the table are of MIPS. Note that these figures assume no-wait-state memory. Hit figures are for the cache used on the 68020.

Since the publication of the results in table a1.1 Motorola have produced a 20 MHz 68020 as a standard item, and for SUN Microsystems a 25 MHz version rated at around 4 MIPS. There is in addition the 8 MIPS 68030 which is not yet in volume production.

Appendix II

Functions Used in the Smalltalk-80 Memory Management Implementation

Introduction.

This is a list of functions and procedures used in the implementation of the Smalltalk-80 system on the Apple Lisa 2/10. Functions are listed by the name used in the code and any equivalent found in Goldberg and Robson is listed by the name used there and by page number in the book "Smalltalk-80 The Language and its Implementation". Procedures that do not appear in the book have their functions discribed. Not all the functions listed in the book are repeated here for the sake of brevity.

address =	given an object it returns the 32 bit address of the first byte (high order) of the first word beyond the two header words of an object.
allocate =	allocate: size odd: oddBit pointer: pointerBit extra:extraWord class: classpointer, pages 668,679,685.
allocatechunk=	Included in function allocate in this implementation, pages 668,684.
build_mem =	Procedure written to set up the memory in the initial state for the load of the Virtual Image.
coreFree =	used to return the number of words of free memory to the interpreter. Used by not defined or indexed.
countDown =	countdown: rootObjectPointer, page 677. Various version used and tested at different points of the implementation. None conformed exactly to the code given. See chapter 6.
countup =	countup: objectPointer, page 677.
deallocate =	deallocate: objectPointer, page 680.

decRefsTo =	decreaseReferencesTo: objectPointer, page 687.
diskGet =	performs the same function as wordGet, only for disk memory. Has a companion function diskPut, but no versions for byte access to disk.
ensure =	used to explicatly cause an object to brought into RAM from a disk segment by the interpreter. Must be called before using the procedure fastfetch.
fastfetch =	same as function fetchword except that in virtual memory implementation it assumes that the object being accessed is in RAM.
fetchBlength =	fetchBytelengthOf: objectPointer, page 687.
fetchByte =	fetchByte: ofObject, page 687.
fetchClassOf =	fetchClassOf: objectPointer, page 687.
fetchFloat =	used to speed transfers of 32 bit IEEE standard reals from memory. Replaced two calls to fetchword, ie. high half, low half.
fetchPointer =	fetchPointer: ofObject, page 686.
fetchWlength =	fetchWordLength: ofObject, page 687.
fetchWord =	fetchWord: ofObject, page 689.
find_to_swap=	see chapter 3, section 3.5
getPointer =	obtainPointer: location, page 670.
HasObject =	used on page 636, but not defined.
HCObyte =	heapChunkOf: objectPointer byte: offset, page 663. It is at this level (routines starting HCO) that the decision between disk and RAM memory access was made for the virtual memory system.
HCObytePut =	heapChunkOf: objectPointer byte: offset put: value, page 663.
HCOword =	heapChunkOf: objectPointer word: offset page 663.
HCOwordPut =	heapChunkOf: objectPointer word: offset put: value page 663.
incRefsTo =	increaseReferencesTo: objectPointer, page 687.
instBclass =	instantiateClass: classPointer withBytes: length, page 687.

InstPclass =	instantiateClass: classPointer withPointers: length, page 687.
InstWclass =	instantiateClass: classPointer withWords: length, page 687.
IntegerObjectOf =	integerObjectOf: value, page 688.
IntegerValueOf =	integerValueOf: objectPointer, page 688.
IsIntObject =	isIntegerObject: objectPointer, page 688.
IsIntValue =	isIntegerValue: valueWord, page 688.
lastPointer =	lastPointerOf: objectPointer, pages 663,685,686. Note that this has been corrected, see Errata in [Xerox83.1]
llstscan =	attemptToAllocateChunkInCurrentSegment: size, page 669. The code used follows this only vaguely.
llstsearch =	attemptToAllocateChunk: size, page 669. In this implementation there were different versions at different times, depending on allocation strategy used (ref: Chapters 4 and 5) none conformed exactly to the specification given.
move_enmass =	used to move large numbers of objects from the RAM memory space to disk memory space. see section 4.10
movesegment =	used to move whole segments out of RAM memory to disk memory.
OOPsFree =	returns the number of free object pointers to the interpreter. Used but not defined or indexed.
seagull =	Procedure used to periodically eat garbage that has accumulated. Garbage being objects that are allocated but with counts = 0.
SignalAtOOPsWordsLeft =	set a semaphore when the amount of free memory falls below a predetermined level.
spaceOccupiedBy =	spaceOccupiedBy: objectpointer, pages 663,680,685. Not used in implementation because extra word not used.
storeByte =	storeByte: byteIndex ofObject: objectPointer withValue: valueByte, page 687.

storePointer =	storePointer: fieldIndex ofObject: objectPointer withValue: valuePointer, page 686.
storeWord =	storeWord: wordIndex ofObject: objectPointer withValue: valueWord, page 686.
wordGet =	RealWordMemory: segment: s word: w, page 656. Used to actually get a word from memory. Functions are also defined for storing a word and on byte sized objects.

Appendix III

Debugging: Methods and Results

Smalltalk-80 is a large system. Operations that on other systems that might be quite trivial can result in a large amount of low level activity. For example, the display of one character in a window pane requires a minimum of 13 calls to the function **fetchWord** with a possible maximum of 26 [Anderson 1985].

The result of this is that debugging statements or other traces can create large amounts of data when the system is run for any significant length of time. The amount of detail that may have to be examined when considering many hundreds of thousand of calls, when one call may produce an incorrect result, is quite daunting.

To this must be added problems introduced by the Lisa's operating system. One effect can be demonstrated by the following example: two runs of the system were made on 14 November 1985 (using a paged virtual memory system, see Section 4.15), the first caused the system to enter the symbolic debugger from a point in the function **object_fault**. A second run caused the the same error to occur, in the same function, but to occur at an earlier point in the program's execution. When the Lisa was rebooted from the external 3.5 inch drive and the hard disk file system "repaired", the apparent bug in the Smalltalk-80 system was removed. From this, it appears that there are some bugs in the Workshop operating system for the Lisa.

Another aberration of the system was observed when using assembler routines. The normal method of writing assembler functions and procedures to be used with Pascal was to have all arguments passed on the stack and, when complete to push the result in the case of a function onto the stack, and then execute a JMP to the location specified by the program counter saved in register A0. An effect observed is that in a complex expression that used many nested assembler routines was that the only result to be returned from these routines was zero, thus resulting in incorrect execution.

The solution used to solve this problem was to break up any complex expressions by using intermediate variables. No reason for the problem could be seen at the time that it was occurring. Using hindsight, it appears that the problem may have been caused by not extending the stack, the zero results comming from the CLR used to create space on the stack for the returned result.

With the nested calls, it appears that the compiler's "fudge factor" of 256 bytes unused on the stack was exceeded. In the assembler routines that were written for this work the LINK and UNLK instructions were not used, neither was the TST instruction that is used to "touch" the stack base to see if there was enough space on the stack. This saved a total of 32 cycles on each call. The fudge factor on the stack was considered to be large enough, as the routines being used at this stage were small and used no stack space except for the passing of parameters. The solution that would have been used, had the identity of the problem been known, was the solution that was used. This is because there were few places where assembler routines were deeply nested in comparison with the number of times that assembler routines were used.

The remaining errors in the system were of two types,

- errors in the design of code, ie. logical errors and
- errors in typing or coding.

The solution to these two problems were more time consuming than dealing with bugs in the operating system or with the stack.

The actual debugging of the memory management code occurred in two stages. The first was to produce what appeared to be a working system from the memory management point of view, but without any of the problems that could be introduced by the interpreter. This was done by erecting scaffolding around the memory manager. A small program was constructed that requested the memory management code to perform its simple functions, such as to allocate and deallocate objects and store and retrieve values to and from objects. After the first stages of hand driven requests for object allocation / deallocation, the program was modified to do this on a semi random basis for size. What became apparent here was the large number of iterations required to create enough objects to completely fill what RAM memory we had allocated to the Smalltalk-80 system and spill over onto the disk based segments.

It became impractical to keep track of results by hand. Code was introduced to perform functions such as checking the consistency of memory:

1- that the total amount of memory claimed by objects in a segment was exactly the same as the amount of space that actually was in a segment. ie. that no extra space had been allocated and that all the segment was accounted for.

2 - that no object considered itself to be of either negative size or of size 0 or size 1. The smallest valid object has a size of at least two words.

3 - that the free object lists did not become circular and that objects in the lists were in fact free.

The major drawback with this method of checking for logical bugs is the amount of time taken to perform the functions mentioned. Checking a disk segment could take quite a large amount of time using random access if the objects contained in it were small. One run was done at an early stage over a length of 500,000 bytecodes. After 24 hours the run was terminated without either producing an error in memory or reaching completion.

The second method used for debugging when the system was running as Smalltalk-80 (or trying to run), was to keep check on objects as they were accessed. In other words, when a value of an object was requested by the interpreter, checks were made to make sure that the objects were allocated and that the word index used was actually inside the objects limits.

Other methods were applied at various times, with varying amounts of success. The most used of these was regression testing [Bently 1985] where the output of one run was compared with previous runs that were thought to be correct and any differences used to locate errors. This method of debugging ran directly into the problem of sheer mass of output. As a result, only selected output could be studied, usually over short ranges of bytecodes, many runs being required to locate points of execution where behaviour differed.

The most successful method of removing bugs once a fully operating Smalltalk-80 system had been built, was to go back to an previous version of the code and remake all the changes made to it, one at a time, until the change in code that resulted in the error was found.

Appendix IV

Results of Memory Management Simulations

The data given here is the raw results of the simulations carried out for the graphs presented in chapter 5. The tables are given in the following format.

- each table represents one method of organising memory.
- the results from all three trace files are shown in each of the tables. The figures are given in a group of 3 for each labeled row / column of the table. The upper figure given is the result from using the trace file **s100**, the middle figure given comes from the trace file **Mscript** and the last figure in each group is from the **factM** trace.

The labels for the rows are,

split at N	== either the value at which a two list system is split or the size of the largest free list that contains only one size of object in a multiple list system.
nodes seen	== the total number of nodes that were examined to fill all requests for memory space.
nodes free	== the number of objects that were found to be free at the end of the simulation run.
mean size free	== the average (mean) size of the free objects at the simulations end.
max. size free	== the size of the largest free object available in the system at the end of the simulation.

split at **N**	5	10	13	15	18	19	20	21
nodes seen	44,563	41,039	35,546	34,348	31,339	31,277	514,347	517,035
	2,547,048	2,513,207	2,512,169	2,477,802	2,410,580	2,416,475	17,225,276	—
	1,164,399	1,091,179	1,087,590	1,075,860	968,864	970,291	2,168,139	21,95,201
nodes free	1,527	1,606	1,681	1,724	1,755	1,763	2,458	2,463
	5,460	5,515	5,517	5,525	6,072	6,072	13,772	—
	1,953	1,949	1,947	1,979	2,155	2,156	3,321	3,321
mean size	92	88	84	82	80	80	57	57
free	19	18	18	18	17	17	7	—
	53	54	54	53	48	48	31	31
max. size	65,536	65,536	65,536	65,536	65,536	65,536	65,536	65,536
free	22,418	22,507	22,507	22,578	22,969	22,969	1,945	—
	65,536	65,536	65,536	65,536	65,536	65,536	65,536	65,536

Results of simulations for a two lists per segment system with 6 segments. Note that the simulation using the trace file Mscript at **N** = 20 did <u>not</u> complete and that there are no results for the trace file Mscript for **N** = 21.

split at N	5	10	13	15	18	20
nodes seen	41,651	33,648	29,734	28,672	23,014	8,658
	2,470,928	2,279,985	2,275,990	2,234,695	1,847,280	160,858
	1,160,296	1,076,787	1,072,107	1,054,550	904,454	104,846
nodes free	1,524	1,501	1,502	1,502	1,443	1,207
	5,279	4,927	4,920	4,859	4,143	623
	1,926	1,842	1,838	1,820	1,657	582
mean size	92	94	94	94	98	117
free	19	21	21	21	25	167
	54	57	57	57	63	181
max. size	65,536	65,536	65,536	65,536	65,536	65,536
free	22,393	22,870	22,870	22,868	22,961	65,536
	65,536	65,536	65,536	65,536	65,536	65,536

Results of simulations using multiple lists per segment (N + 1 lists), 6 segments were used.

split at N	5	10	15	20	30	40
nodes seen	39,095	28,275	22,097	8,071	7,952	7,922
	3,427,787	2,976,254	2,814,729	108,914	109,242	104,618
	1,055,291	912,743	867,141	37,009	27,985	21,451
nodes free	1,599	1,566	1,569	1,167	1,168	1,139
	4,391	3,963	3,887	344	446	166
	1,854	1,737	1,698	385	351	285
mean size	88	90	90	121	121	124
free	23	26	26	304	234	630
	56	60	62	273	300	369
max. size	65,536	65,536	65,536	65,536	65,536	65,536
free	33,,498	33,969	34,088	65,536	65,536	65,536
	65,536	65,536	65,536	65,536	65,536	65,536

Results of simulations for the multiple lists in a linear memory system with the equivalent of 6 segments memory (6 * 128k bytes).

split at N	5	10	15	20	30	40
nodes seen	41,651	33,648	28,672	8,658	8,486	8,209
	2,470,928	2,279,985	2,234,695	160,858	161,403	107,500
	1,160,296	1,076,787	1,054,550	104,864	98,100	22,621
nodes free	1,524	1,501	1,502	1,207	1,206	1,189
	5,279	4,927	4,859	623	623	333
	1,926	1,844	1,820	582	580	414
mean size	92	94	94	117	117	119
free	19	21	21	167	167	314
	54	57	57	181	181	254
max. size	65,536	65,536	65,536	65,536	65,536	65,536
free	22,393	22,870	22,868	65,536	65,536	65,536
	65,536	65,536	65,536	65,536	65,536	65,536

Results of simulations for the multiple lists per segment system, 6 segments and a modified segment changing algorithm are used.

Appendix V

Implementation of Reference Decrimenting Routines

```
PROCEDURE newCountDown(root : integer);
  LABEL
      13,666,999;
  VAR
      list,curnt,ctemp,offset : integer;

  PROCEDURE link(anObj : integer);
      BEGIN
      IF list < 0 THEN
          BEGIN
          classPut(anObj, -1);          { if there is nothing in the list then object }
          list := anObj;                { anObj is the first in the list. }
          END
      ELSE
          BEGIN
          classPut(anObj, list);        { otherwise object anObj is added to the }
          list := anObj;                { start of the LIFO queue }
          END;
      END;

  BEGIN
  here := -1;                           { there is no object we are currently scanning }
  list := -1;                           { there is no list of objects the we have to scan }
  offset := 0;

  13:
  IF root >= 0 THEN
      IF lowerCount(root) = 0 THEN BEGIN
          ctemp := classGet(root);
          IF lastPtr(root) > 2 THEN        { the either pointer object or }
              BEGIN                        { a method so must be enqueued }
              link(root);
              root := ctemp;
              goto 13;
              END
          ELSE
              BEGIN
              deallocate(root);         { the object can be deallocated as no pointers }
              root := ctemp;
              goto 13;                  { class pointer now root object, tail recursion }
              END;
          END;
```

```
666:
IF (list < 0) and (here < 0) THEN goto 999;          { see if any more obects to scan }
IF here < 0 THEN
    BEGIN
    here := list;
    OTjunk := classGet(here);
    offset := LastPointer(here);
    END;

offset := offset - 1;
IF offset = 1 THEN                              { if object completely scanned then deallocate }
    BEGIN
    deallocate(here);
    here := -1;
    goto 666;                                   { tail recursive call to countDownAll }
    END;

root := wordGet(here, offset);                  { get the next pointer and use it as root }
goto 13;                                        { tail recursive call to the countDown procedure }

999:
END;
```

Algorithm 6.1: Pascal implementation of Krasner,McCullogh method with modifications.

```
PROCEDURE countDown (rootObj : integer);
    VAR
        tobj, count, offset : integer;
    BEGIN
    IF rootObj >= 0 THEN    { if object is NOT a small integer }
        BEGIN
        count := lastpointer(rootObj);

        IF lowerCount(rootObj) = 0 THEN
            BEGIN
            FOR offset := 1 TO (count - 1) DO
                BEGIN
                tobj := wordGet(rootObj, offset);
                IF tobj >= 0 THEN
                    IF lowerCount(tobj) = 0 THEN OTjunk := addLink(tobj);
                END;
            deallocate(rootObj);
            END;
        END;
    END;
```

Algorithm 6.2: The countDown procedure used in the delayed deallocation of secondary objects. The function **addLink** add the pointer **tobj** to the linked list of objects that are to be deallocated by **seagull**. This link is pointer to by the global variable **OTjunk**.

```
PROCEDURE seaGull;
    VAR
        here, next : integer;
    BEGIN
    IF Otjunk > 0 THEN BEGIN                    { if there are nay objects to be removed }
        BEGIN                                   { the first object in the chain }
        here := Otjunk;
        OTjunk := NonPointer;
        next := getLink(here);

        WHILE here > 0 DO                       { there are objects to be deallocated. }
            BEGIN
            countDown(here);
            DisLink(here);
            here := next;
            IF next > 0 THEN next := getLink(next);
            END;
        END;
    END;
```

Algorithm 6.3: The procedure used to remove objects that have been noted as secondary objects to be deleted. The assembler routine **DisLink** removes the node that was passed to **countDown** from the linked list. Note that **OTjunk** points to a new linked list of objects to be counted down at the end of the procedure. **GetLink** get the value of the next link in the chain.

```
; the following are the A5 relative locations where the global variables that contain either
; address to large data structures such as the 32 bit location table for objects or global
; variables such as OTjunk.

FRTURN   .EQU   -$E4     ; so we can return data apart from function values.
NADDR    .EQU   -$E0     ; address of 41 element array that has the free lists.
SADDR    .EQU   -$DC     ; not used.
LADDR    .EQU   -$D8     ; address of 32 bit array of location bits.
CADDR    .EQU   -$D4     ; address of  8 bit array of count bits.
BADDR    .EQU   -$D0     ; address of  8 bit array of flag bits.
KADDR    .EQU   -$CC     ; address of 16 bit array of objects to kill.
OTJUNK   .EQU   -$04     ; the variable OTjunk, offset from A5.
CFREE    .EQU   -$EC     ; the variable core_free, offset from A5.
          ;
        .PROC      CNTD3
        MOVE.L     (A7)+, A0           ; return address.
        MOVE.W     (A7)+, D0           ; root object pointer.
        ;
        MOVEM.L    D3/A2-A3, -(A7)     ; save results on the stack.
        AND.L      #$7FFF, D0
        MOVE.L     D0, -(A7)           ; need this value to deallocate at the end.
        MOVE.L     D0, D2
        ;
        MOVE.L     LADDR(A5), A1
        LSL.L      #2, D0              ; * 4 as long words byte addressed.
        MOVE.L     0(A1,D0.L), A1      ; address of the root object in mem.
        ;
        ;******** GET THE LAST POINTER OF THE ROOT  OBJECT.
        ;
        CLR.L      D1
        MOVE.L     BADDR(A5), A2
        MOVE.B     0(A2,D2.W), D0      ; d2 has the value of d0.
        ANDI.B     #$2, D0             ; test the pointer bit.
        BEQ        NOTPTR
        MOVE.W     (A1), D1            ; return the size bits.
        BRA        LSTART
        ;
NOTPTR:                                ; but could be a method.
        CMPI.W     #11, $2(A1)         ; see if is a method.
        BNE        NOTMETH
        MOVE.W     $4(A1), D1
        ANDI.W     #$3F, D1
        ADDI.W     #3, D1              ; add in header_size + 1.
        BRA        LSTART
        ;
NOTMETH:
        MOVE.W     #2, D1
        ;
        ;********
        ;
LSTART:
        MOVE.L     CADDR(A5), A2
        MOVE.L     KADDR(A5), A3
        ;
        ;******** THE MAIN  LOOP  STARTS  HERE.
        ;
        CLR.W      D0
        MOVE.W     OTJUNK(A5), D0      ; save a copy of the first link which doesn't change.
LOOP1:
        ADDQ.L     #2, A1              ; first time around points to class bits.
        SUBQ.L     #1, D1
        BLE         CNTEND
        ;
        ;********  GET WORD.
        ;
        MOVE.W      (A1), D2            ; also does the test for  less than.
        ;CMPI.W     #27, D2             ; objs 0..26 count bits of 255 => can't be counted
down.
        BLT         LOOP1
        ;
        ;********  LOWER THE COUNT OF THE POINTER JUST FOUND.
        ;
        MOVE.B      0(A2,D2.W), D3      ; get the pointers count bits.
        ;
        CMPI.B      #$FF, D3            ; check to see if we have overflowed.
        BEQ         LOOP1               ; if so, cant count down as it has reached the stick
point.
        SUBQ.B      #1, D3
        MOVE.B      D3, 0(A2,D2.W)      ; else store the value, does test for 0 at same time.
        BGT         LOOP1
        ;
        ;********  ADD A LINK TO THE KILL LIST.
        ;
ADDLINK:
        TST.W       D0
        BLT         FIRSTP              ; -1 = $FFFF => the sign bit is still set !
        ADD.L       D0, D0              ; * 2.
        AND.L       #$FFFF, D2          ; so we can use d2 as a long.
        MOVE.W      0(A3,D0.L), D3      ; get next link of list.
        MOVE.W      D2, 0(A3,D0.L)      ; replace the link of the list.
        ADD.L       D2, D2
        MOVE.W      D3, 0(A3,D2.L)      ; new second link points to old second link.
        LSR.L       #1, D0
        BRA         LOOP1
        ;
FIRSTP:
        CLR.L       D0
        MOVE.W      D2, D0
        MOVE.W      D0, OTJUNK(A5)
        ADD.L       D0, D0
        MOVE.W      #-1, 0(A3,D0.L)
        LSR.L       #1, D0
        BRA         LOOP1
        ;
        ;***  DEALLOCATE THE ROOT OBJECT  **********
        ;
CNTEND:
        MOVE.L      (A7)+, D0           ; get the value of the root object off of the stack.
        MOVE.L      BADDR(A5), A1
        ORI.B       #$4, 0(A1,D0.W)     ; set the  F bit to show that the object is free.
        ;
        MOVE.L      LADDR(A5), A1       ; address of the array of location bits.
        MOVE.L      D0, D1
        LSL.L       #2,D1               ; *4 for long word offset into the array.
        MOVE.L      0(A1,D1.L), A1      ; the address of the object, and of the size bits field.
        MOVE.W      (A1), D2            ; get the size bits of the object being deallocated.
        AND.L       #$FFFF, D2          ; set top 2 bytes are 0 so can d0 the long compare.
        ;
        MOVE.L      CFREE(A5), D3       ; get the value of the core free variable.
        ADD.L       D2, D3
        MOVE.L      D3, CFREE(A5)       ; core free has been updated.
        ;
        CMP.L       #40, D2             ; size = 40 is the largest object with a single list.
        BGT         ONBIGL              ; if size > 40 then it goes on the big list.
        MOVE.L      NADDR(A5), A2       ; a2 = address of the array of lists.
```

```
          ADD.W       D2, D2              ; offset is *2 as word size cells in array.
          MOVE.W      0(A2,D2.W),D3       ; store the value of the old head of free list d1/2.
          MOVE.W       D0,0(A2,D2.W)      ; head of free list points to the root object.
          MOVE.W      D3,$2(A1)           ; class bits of 1st object now point to the 2nd object.
          ;
          BRA         THEEND
ONBIGL:
          MOVE.L      NADDR(A5), A2       ; a2 = address of the array of lists.
          MOVE.W      #82,D2              ; offset is *2 as word size cells in array: as max = 41.
          MOVE.W      0(A2,D2.W),D3       ; store the value of the old head of free list d1/2.
          MOVE.W      D0,0(A2,D2.W)       ; head of free list points to the root object.
          MOVE.W       D3,$2(A1)          ; class bits of 1st object now point to the 2nd object.
          ;
          ;*********
          ;
          ;
THEEND:
          MOVEM.L     (A7)+, D3/A2-A3
          JMP         (A0)

          .END
```

Algorithm 6.4: The assembler implementation of the procedure **countDown** as described in section 6.5. Note that the Lisa Pascal compiler uses registers A2-A7 and D3-D7 so the values that they contain must be saved and restored.

APPENDIX VI

Program for Conversion of the Xerox Virtual Image

This program was used to convert the virtual image received on a 9 track tape from Xerox from the standard format to the format described in chapter 3. The initial input to the program a single file of type integer. This file contains both the objects in the virtual image and their ROT table entries. The program is semi interactive in that it can be halted at the end of each phase. With the 1.5 mega bytes now avaliable on the Lisa this could be re-written so that main memory is used in place of temporary disk files. This would result in a large increase in execution speed. This was not done as this program was only ever run once after the memory had been upgraded.

There are three major sections of code, they are

1. strip the initial 512 bytes from the file, read the remainder of the objects from the file and write them to a temporary file.
2. decode the ROT (after the objects in the file) data and write it to a file.
3. convert the objects in the temporary file of step 1 to the new format using information from the output file of step 2 and write this to another file.

```
PROGRAM convertOPPs(input, output, tmp, OPPs ,new_OPP, newROT);
CONST
    mContext  =11;
    bContext  =12;
    method    =17;          { compiledMethods are pointers,integers,and bytecodes }
                            { note this is the converted value (div 2) }
    ObjSpace  = 517760;
    ROTspace  = 77472;  { one segment }
    headersize = 2;         { two words of header size and class }

TYPE
    ptr = ^node;
    node = record
        kind, count : integer;
        next        : ptr;
        end;

    inp_file = file of integer;

    t_ROT = record
        count : integer;
        bits  : packed array [1..4] of boolean;
        end;

    ROT_file = file of t_ROT;

VAR
    which, i         : integer;
    page_break       : 1..256;          { the number    of words in a page }
    byte_count       : longint;         { # of bytes        read }
    object_count     : longint;         { # of objects that have been seen }
    tmp,
    new_OPP,
    OPPs             : inp_file;
    newROT           : ROT_file;

{*****************************************************************}

FUNCTION BitAndM (int1, int2 : integer):integer;EXTERNAL;  { i1 and i2 }

FUNCTION BShiftR (int1, int2 : integer):integer;EXTERNAL;   { shift i1 right by i2 }

FUNCTION BitBlast (int : integer):integer;EXTERNAL;  { to decode method headers }
```

```
{*****************************************************************}

PROCEDURE quitter;
    VAR
        ans : char;
    BEGIN
    write (output, ' HALT ? [y,n] : ');
    readln (input, ans);
    IF ans in ['Y','y'] THEN halt;
    END;

{*****************************************************************}

PROCEDURE strip(var OPPs : inp_file);
    { is used to strip off the first 512 bytes of the file which are just header information }
    VAR
        i, word, tp  : integer;
    BEGIN
    writeln (output, '  ** the first ten bytes of the file in r10');
    byte_count := 0;
    FOR i := 1 TO 256 DO
        BEGIN
        word := OPPs^;
        byte_count := byte_count + 2;
        get (OPPs);
        IF i <= 5 THEN
            BEGIN
            { tp := getbyte (word, 0);  write (output, 'i = ', i:4, '   ', tp:5); }
            { tp := getbyte (word, 1);  writeln (output, '    ', tp:5); }
            END
        ELSE IF word <> 0 THEN
            writeln (output, ' word = ', i, ' is = ', word, ' NOT 0');
    END;
END;

{*****************************************************************}

PROCEDURE skip_objects(var ifile, ofile : inp_file);

    { write the object table to another file so we can get at the ROT, by dumping }
    { the file on the VAX I was able to see where there was a block of zeros }
    { marking the end of the OPPs and the start of the ROT. They conform to  }
    { calculations done on the length of sections. (first 8 bytes) }

    VAR
        size, i         : integer;
        length, j       : longint;
    BEGIN
    length := 0;
    j := 0;
    WHILE byte_count < (objSpace + 512) DO
        BEGIN
        j:= j + 2;;
        size := ifile^;
        get (ifile);
        byte_count := byte_count + 2;
        length := length + 1;
        ofile^ := size;
        IF (j < 20) or (j > (objSpace - 20)) THEN writeln (output, (j div 2), size);
        put (ofile);
        END;

    writeln (output, ' length of OOPs in words ', length:7, ' in bytes ', (length*2):7);
    writeln (output, ' total bytes read to end of OOPs ', byte_count);

    size := ifile^;
    WHILE size = 0 DO
        BEGIN
        get (ifile);
        byte_count := byte_count + 2;
        size := ifile^;
        END;

END;

{************************************************************}

PROCEDURE decode_ROT(var ifile : inp_file; var nROT  : ROT_file);
    { decode the ROT as given on the disk so that it can be used easily }
    VAR
        i            : integer;
        rec          : t_ROT;
        shifted, j   : longint;
    BEGIN
        writeln (output, 'bytes read so far = ', byte_count);
        i := ifile^;
        writeln (output, 'looking at first word of ROT = ', i);

        shifted := 0;  j := 0;
        WHILE (shifted < ROTspace) DO
            BEGIN
            i:= ifile^;
            get (ifile);
            j:= j + 1;

            rec.count := BshiftR (BitAndM (i, $FF00), 8);
            rec.bits[1] := BitAndM (i, $0080) <> 0;  { O bit  }
            rec.bits[2] := bitAndM (i, $0040) <> 0;  { P bit  }
            rec.bits[3] := bitAndM (i, $0020) <> 0;  { F bit  }
            rec.bits[4] := false;                    { unused }
            get (ifile);                             { skip location bits as not used }

            byte_count := byte_count + 4;
            shifted := shifted + 4;
            nROT^ := rec;
            put (nROT);
            END;

    writeln (output,'bytes at end of decode ROT = ',byte_count,' shifted ',shifted);
END;

{************************************************************}

PROCEDURE translateOPPs(var OPPin : inp_file;
            var nROT  : ROT_file;
            var nOPP  : inp_file);
    VAR
        bytes_seen                 : longint;
        class_keep,
        class, size, lits, i       : integer;
        Hsize, Hcomp ,count        : integer;       { check to see we don't goof }
        objNum                     :  integer;
        rec                        : t_ROT;
        kind                       : boolean;
    BEGIN
    bytes_seen := 0;
```

```
get (nROT);                                  { skip the pointer for object 0 }

objNum := 2;
size := OPPin^;
WHILE (bytes_seen < objSpace) DO
    BEGIN
    writeln (output, ' >>> ', bytes_seen);
    REPEAT
        rec := nROT^;
        get (nROT);                          { get the ROT entry }
        UNTIL not (rec.bits[3]);

    size := OPPin^;
    get (OPPin);
    nOPP^ := size;
    put (nOPP);
    class := OPPin^;
    get (OPPin);
    class := bitblast (class);
    class_keep := class;
    nOPP^ := class;
    put (nOPP);
    bytes_seen := bytes_seen + 4;
    count := 2;
    Hsize := size;

    IF class = method THEN
        BEGIN                        { mixture of byte codes and pointers }
        class := OPPin^;             { actually header }
        get (OPPin);
        kind := true;
        bytes_seen := bytes_seen + 2;
        count := count + 1;
        Hcomp := class;
        class := BitAndM (class, 126); { here same as in Blue Book }
        lits  := BShiftR (class, 1);
        class := bitblast (Hcomp);
        nOPP^ := class;
        put (nOPP);
        FOR i := 1 TO lits DO
            BEGIN
            class := OPPin^;
            get (OPPin);
            class := bitblast (class);
            nOPP^ := class;
            put (nOPP);
            bytes_seen := bytes_seen + 2;
            count := count + 1;
            END;
        FOR i := 1 TO (size - headersize - 1 - lits) DO   { -1 for extra word in H }
            BEGIN
            class := OPPin^;
            get (OPPin);
            nOPP^ := class;
            put (nOPP);
            bytes_seen := bytes_seen + 2;
            count := count + 1;
            END;
        END
        ELSE FOR i := 1 to (size - headersize) DO
            BEGIN
            kind := false;
            class := OPPin^;
```

```
                    get(OPPin);
                    IF (rec.bits[2]) or (class_keep = bContext) or
                         (class_keep = mContext) THE class := bitblast (class);  { is a pointer }
                    nOPP^ := class;
                    put (nOPP);
                    bytes_seen := bytes_seen + 2;
                    count := count + 1;
               END;
               IF count <> size THEN
                    BEGIN
                    writeln (output,' * ERROR * size & count not equal compMeth is ',kind);
                    IF kind THEN
                         writeln (output,'size head lits obj bytes', Hsize, Hcomp, lits, objNum,
                         bytes_seen);
                    halt;
                    END;
               size := OPPin^;
               objNum := objNum + 2;
               END;
END;
```

```
{* MAIN PROGRAM ************************************************}

    BEGIN
    byte_count := 0;
    which := 1;
    WHILE which <> 0 DO
        BEGIN
        FOR i := 1 TO 30 DO writeln (output,' ');
        gotoxy (0,0);
        writeln (output, '  >> what to do ?');
        writeln (output, '     0/ quit');
        writeln (output, '     1/ remove the OOPs from the file');
        writeln (output, '     2/ remove the ROT after 2');
        writeln (output, '     3/ translate OOPs');
        writeln (output, '');
        write (output, '      ===> which ? : ');
        readln (input, which);

        CASE which OF
            1 : BEGIN
                reset (OPPS, 'st/image.int');
                strip (OPPs);
                rewrite (tmp, 'me/OPPS.tmp.int');
                skip_objects (OPPs, tmp);
                close (tmp, lock);
                END;
            2 : BEGIN
                rewrite (newROT, 'st/virROT.ROT');
                decode_ROT (OPPS, newROT);
                close (newROT, lock);
                close (OPPS, lock);
                END;
            3 : BEGIN
                reset (tmp, 'me/OPPS.tmp.int');             {has the OPPs in it }
                reset (newROT, 'st/virROT.ROT');            { has the ROT }
                rewrite (new_OPP, 'st/virOPP.OPP');         { converted OPPs will go here }
                translateOPPs (tmp, newROT, new_OPP);
                close (new_OPP, lock);
                close (newROT, lock);
                END;
            END;        { of case }

            write (output, ' ************* OK ? : ');
            readln (input);
        END;  { while }

    END.
```

References

ANDERSON, Philip David; 1986, personal communication.

APPLE; 1986, The Smalltalk Group APPLE Computer; The Smalltalk-80 Programming System for the Macintosh. July 1986, pg 10.

BALLARD, Stoney; SHIRRON, Stephen; 1983, The Design and Implementation of VAX/Smalltalk-80. In KRASNER, Glen. Smalltalk-80 Bits of History, Words of Advice. Addison-Wesley 1983 P. 127-150.

BELL, C. Gordon; 1986, RISC: Back to the Future?. Datamation Vol 32 #11 June 1,1986. pg 96-108.

BENTLY, Jon; 1985, Bumper-Sticker Computer Science. Communications of the ACM Vol 28 #9 September 1985 pg 896.

CARNEVALE, Ted; 1985, C to Pascal. Byte February 1985 pg 139-144.

COHEN, Jacques; 1981, Garbage Collection of linked Data Structures. Computing Surveys Vol 13 #3 Septemeber 1981 pg .

DENNING, Peter J.; 1968, The Working Set Model of Program Behaviour. Communications of The ACM Vol 11 #5 March 1968 pg326.

DENNING, Peter J.; 1970, Virtual Memory. Computing Surveys Vol 2 #3 September 1970 pg 178.

DUFF, Charles B.; 1986, Designing an Efficient Language. Byte Vol 11 #8 August 1986 pg 211-224.

FORSMAN, Fred; 1983, Pascal Code Cruncher's Handbook. In Lisa ToolKit 3.0 vol. 2, 1983.

GOLDBERG, Adele; ROBSON, David; Smalltalk-80 the Language and its Implementation. Addison-Wesley 1983a.

GOLDBERG, Adele; ROBSON, David; 1983b ibid. pg 678.

GOLDBERG, Adele; ROBSON, David; 1983c ibid. pg 668-669.

GOLDBERG, Adele; ROBSON, David; 1983d ibid. pg 671-674.

HWANG, Kai; BRIGGS, Fay A; 1984, Computer Architecture and Parralel Processing. McGraw-Hill 1984. page 71-72.

JAULENT, Patrick; 1985, The 68000 Harware and Software. MacMillan Publishers Ltd 1985. page 4.

KAEHLER, Ted; KRASNER, Glen; 1983, LOOM-Large Object-Oriented Memory for Smalltalk-80 Systems. In Krasner, Glen. Smalltalk-80 Bits of History, Words of Advice. Addison-Esley 1983 pg 251.

KEARNS, J.P; DeFAZIO, S; 1983, Locality of Reference in hierarchical Database Systems. IEEE Transactions on Software Engineering, vol SE-9 #2 March 1983 pg 128-134.

KING, Tim; KNIGHT, Brian; 1983, Programming the M68000. Addison-Wesley 1983.

KRASNER, Glen; McCULLOUGH, Paul; 1984, A Better Freeing Algorithm. In Smalltalk-80 News Letter #3, May 1984

KNUTH, Donald E.; The Art of Computer Programming, Vol. 1 Fundamental Algorithms. Addison-Wesley 1973a pg 436.

KNUTH, Donald E.; 1973b ibid. pg 448.

KNUTH, Donald E.; 1973c ibid. pg 413.

MACLEAN, M.A.; 1985, personal communication.

McCULLOUGH, Paul L.; 1983, Implementing the Smalltalk-80 System: The Tektronix Experience. In Krasner, Glen. Smalltalk-80 Bits of History, Words of Advice. Addison-Esley 1983 pg 59-77.

MACGREGOR, Doug; RUBINSTEIN, Jon; 1985, A Performance Analysis of MC68020-based Systems. IEEE Micro, vol 5 #6 December 1985.

MORGAN, Cris; 1985, An Interview with Wayne Rosing, Bruce Daniels, and Larry tesler A Behind the Scenes Look at the Development of Apple's Lisa. Byte. vol 8 #2 Febuary 1985.

OLDEHOEFT, Rodney R.; ALLAN, Stephen J.; 1985, Adaptive Exact-Fit Storage Management. Communications of the ACM. vol 28 #5 May 1985. pg 506-511.

ORGANIK, Elliott I; 1973, Computer System Organization: The B5700/B6700 Series. Academic Press 1973.

PARNAS, David; 1986, personal communication.

RORIGUEZ-ROSELL, Juan; 1976, Empirical Data Reference Behaviour in Data Base Systems. IEEE Computer vol 9 #11November 1976 pg 9-13.

THACKER, C.P et al.; 1982, Alto: APersonal Computer. In Bell, Daniel P. Gordon; Newell, Allen; Computer Structures: Principles and Examples McGraw-Hill 1982.

UNGAR, David M; PATTERSON, David A; 1983, Berkeley Smalltalk: who knows where the time goes. In KRASNER, Glen. Smalltalk-80 Bits of History, Words of Advice. Addison-Wesley 1983 pg 193.

WIRFS-BROCK, Allen ;1983a, Design Decisions for Smalltalk-80 Implementors. In KRASNER, Glen. Smalltalk-80 Bits of History, Words of Advice. Addison-Wesley 1983 pg 47.

WIRFS-BROCK, Allen; 1982b ibid. pg 57.

WYVILL, Brian; 1983, personal communication, 1986.

XEROX; 1983a, Smalltalk-80 Virtual Image Version 2. Software Concepts Group Xerox Palo Alto Research Center. 1983.

XEROX; 1983b ibid. pg 3.