COMPUTER SYSTEMS WITH A VERY LARGE ADDRESS SPACE

AND

GARBAGE COLLECTION

by

PETER B. BISHOP

B.S., Massachusetts Institute of Technology
(1972)

S.M., Massachusetts Institute of Technology
(1972)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(MAY 1977)

Signature redacted

Signature of Author ....
Department of Electrical Engineering and Computer Science, May 5, 1977

Signature redacted

Certified by ....
Thesis Supervisor

Signature redacted

Accepted by ....
Chairman, Department Committee

JUN 29 1977
LIBRARIES

COMPUTER SYSTEMS WITH A VERY LARGE ADDRESS SPACE AND GARBAGE COLLECTION

by

PETER B. BISHOP

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 1977 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

ABSTRACT

The concept of objects is beginning to gain acceptance throughout the field of computer science. A new computer system is proposed that provides hardware support for objects and object references that can be used in all applications of objects. The new system provides small object references that can be copied freely, makes very small objects efficient, and retrieves the storage for inaccessible objects automatically. This system is compared with some widely used existing systems, and while its speed seems to be competitive, it is much easier to use.

Object references provide protection in the new system as do capabilities in capability systems. The object reference in the new system contains an address from a linear, paged virtual address space rather than a unique ID. Use of small objects is made feasible by efficiently grouping objects into areas. Objects in the same area may be placed on the same page. The system automatically and efficiently maintains lists of inter-area links that allow single areas to be garbage collected independently of the rest of the system. The garbage collector can determine whether objects have been inappropriately placed in an area and can move these objects to more appropriate areas automatically.

Thesis Supervisor: Carl Hewitt, Associate Professor

Acknowledgements

I would like to thank my thesis supervisor, Prof. Carl Hewitt, for providing an environment at M.I.T. in which I could pursue my research. My work with Prof. Hewitt on Planner gave me the detailed knowledge of how to design and implement very advanced programming languages that enabled me to see clearly the shortcomings of current computer systems. I would also like to thank M.I.T. for constructing the Multics computer system and for making it available to students through the Student Information Processing Board. My experience in programming on Multics enabled me to arrive at my somewhat unusual view of computer systems and computer system design that resulted in this thesis. I would like to thank David D. Redell for patiently talking with me and for teaching me most of what I know about capability systems. I would also like to thank Prof. Liba Svobodova for her encouragement toward the end of the work on my thesis. She also did a superb job of reading the thesis. Thanks are also due to my other readers, Prof. Vaughann Pratt, Michael D. Schroeder, and Prof. Michael Dertouzos, for their helpful comments. I would like to thank my family for their constant support and encouragement, without which I would not have been able to continue the struggle for seven years. Finally, I owe a special debt of gratitude to my wife, Catherine W. Bishop, not only for her love, eternal patience, and support, but also for her financial support at jobs that were not always pleasant.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0522.

Table of Contents

1. Introduction 7
1.1 Objects 7
1.2 Widespread Acceptance of Objects 9
1.3 Object References 10
1.4 Are Object References Unstructured? 14
1.5 Computer System Design 15
1.6 Contributions of the Thesis 18
1.7 Organization of the Thesis 21

2. Computer System Design: An Historical Perspective 22
2.1 Size of the Address Space 24
2.1.1 Small Address Space Virtual Memory (SAV) Systems 24
2.1.2 Large Address Space Virtual Memory (LAV) Systems 25
2.2 Single Address Space 26
2.3 Capability Systems 28
2.3.1 Direct Capability (DC) Systems 34
2.3.2 CUID Systems 35
2.3.3 Garbage Collection 37
2.3.4 Small Objects 41

3. The New System - ORSLA 43
3.1 Address Space 43
3.2 Words 50
3.3 Enforcing Restrictions on the Use of Object References 52
3.4 Enforcing the High Level Restrictions 57
3.5 Format of the Object Reference 60
3.6 Monitoring 62
3.7 Small Objects 67
3.8 Garbage Collection 69

4. Garbage Collection in Areas 70
4.1 Overview of Garbage Collection 70
4.2 Copying Garbage Collection 71
4.3 Large Garbage Collections 75
4.4 Other Approaches to Large Garbage Collections 78
4.5 Inter-Area Links 80
4.6 Cables 84
4.7 Local Computation Areas 89
4.8 Using Inter-area Links in Machine Language Programs 91
4.9 The Area Object 93
4.10 Garbage Collection on ORSLA 96
4.11 Multiple Simultaneous Garbage Collections 105
4.12 Multiple-Area Cycles 108

4.13 The Effect of Errors in the Garbage Collector 111
4.14 Other Related Work 111

5. Maintaining the Lists of Inter-area Links 116
5.1 Creation of Inter-Area References - What Must Be Computed 116
5.2 Computing Checks for Load and Store Operations Efficiently 123
5.3 Special Hardware Needed on ORSLA 129
5.3.1 Virtual Memory Mapping 129
5.3.2 Page Map 131
5.3.3 Cache 137
5.4 Comparison of ORSLA with Other Systems 142

6. Placement of Objects in Areas 154
6.1 Initial Placement 154
6.2 Directories 157
6.2.1 List of Named Objects 159
6.2.2 Garbage Collecting Areas 161
6.2.3 Deleting Areas 163
6.3 The Mover 166
6.4 Garbage Collection Revisited 168
6.5 Problems Solved by the Automatic Mover 174
6.5.1 Multiple-Area Cycles 178
6.6 Problems Created by the Mover 184
6.7 Costs of Garbage Collection 186
6.8 Garbage Collection Corresponds to Other Operations on SAV Systems ... 195
6.9 Comparison of Inter-area Links with Linking on SAV and LAV Systems . 197

7. Protection 201
7.1 Enforcing the Size of Objects 203
7.2 Abstract Objects 212
7.2.1 The Lower Operation 214
7.2.2 Elevate Operations 217
7.2.3 Access Control Field 219
7.3 Domains 228
7.4 Revocation 232
7.5 Allocation of Storage 241
7.5.1 First Free Storage Data Type 243
7.5.2 Allocating Address Space and Storage to Areas 246
7.5.3 Second Free Storage Data Type 249
7.5.4 Third Free Storage Data Type 250
7.5.5 Conclusion 252

Appendix A. The Size of the Address Space 254
Appendix B. The Area Object 258
References 261

Chapter 1
Introduction

The first electronic computer was designed in 1945 just over thirty years ago. Since then the electronics industry has advanced at a breakneck rate to make electronic components as ideal for building computers as possible. Reliability and speed have increased while power consumption and cost have decreased. At one time the design of CPUs was largely influenced by a need to minimize costs for the CPU and the memory. A processor and its memory, however, are just modules in a larger system, the computer system, that must interact in a meaningful way with the outside world. Other elements of this system are the system software, I/O devices, and user software. Today, the cost of the CPU is a relatively minor part of the cost of a computer system. Therefore the design of the CPU should maximize the efficiency of the rest of the computer system rather than minimize the cost of the CPU.

Some computers are used for controlling machinery. If the machinery is not too complex, the programs needed are rather simple. The current state of the art in computer system design is adequate for the needs of such systems. Some computer systems, however, are used as general purpose tools for processing information. Such systems are currently so unreliable and difficult to use that they do not achieve their full potential as tools for processing information. To achieve this potential, the design of the CPU should be affected by the fundamental requirements of computing. Unfortunately, there has not been an adequate understanding of what these fundamental requirements are.

1.1 Objects

In the last few years, however, the concept of *objects* has emerged and seems to be gaining acceptance throughout the field of computing. Computations are performed on objects. Each object is a model of an ideal object known intuitively to the programmer. If the programmer can create new kinds of objects, then the programmer can write programs that deal with the objects with which he is familiar. When a computer does not permit the set of objects to be extended then the programmer must continuously translate the algorithms in his mind so they will make use of the small set of objects that are provided by

the computer. Often the concept of objects is only supported by a compiler or an interpreter and not by the operating system or the hardware. If the entire computer system is involved in supporting objects, then a more hospitable environment can be provided for subsystems that use objects. Furthermore, it may be possible to identify several utility operations that can be performed by the computer system for all of its subsystems rather than forcing each subsystem to struggle with these tasks separately.

This thesis presents a computer system that supports objects in the hardware and in the operating system as well as in interpreters and compilers. This computer system is a logical successor to several existing computer systems on which objects are supported in different ways. The new system is carefully designed to improve reliability and make the system easier to use without sacrificing speed. This chapter discusses the concept of objects, its origins, and some of the techniques that have been used to support it. The most important mechanism for supporting objects in hardware is the object reference. The discussion of objects in this chapter concludes with a list of design criteria that must be met by the mechanism for object references that is provided by the system.

In order to implement an abstract object, the programmer first formalizes his intuitive knowledge of the object by creating an *abstraction* of the ideal object, i.e. by defining the *behavior* of a set of abstract operations on the object. A set of *operations* is then written for the object in the computer. The behavior of these operations defines the abstraction that has been implemented. The computer system will provide an initial set of generally useful objects that will include integers, character strings, procedures, arrays, lists, file directories, etc. In order to implement an abstraction each object must also have a *representation*. Some of the operations on an object will be implemented in terms of manipulation of the representation of the object while some operations will be implemented in terms of the abstract operations that have been defined on the object. In order for any operation to be able to manipulate an object, however, it must be able to refer to the object. This is achieved by the *object reference*. It should be easy to copy and move an object reference regardless of the size of the object it references. Thus the object reference usually contains a pointer to the representation of the object.

1.2 Widespread Acceptance of Objects

The concept of objects in computing appeared first in Simula 67 [Birtwistle73] in order to provide a good conceptual basis for defining and manipulating the objects involved in simulations. Since then the concept has gained increasing acceptance in the field of programming languages. As with all good conceptual foundations, it is possible to look at programming languages defined earlier than Simula 67 and identify the objects in these languages as well. The concept of objects has also gained acceptance in the field of operating systems and is emerging in the field of data base management systems. All three fields have independently developed their own concept of objects, thus suggesting that objects are fundamental to computing.

The underlying foundation of the concept of objects can be found in mathematics. Mathematicians define mathematical objects by specifying the behavior of the operations on the objects. This corresponds to an implementation independent definition of an object. Computing has merely expanded this approach to encourage computable definitions. If it were possible to create definitions of objects that could be used efficiently directly from mathematicians' definitions, then this would be preferable.

The concept of objects is important in the field of operating systems. Dijkstra introduced the concept of levels of abstraction in the T.H.E. system [Dijkstra68]. This concept has a wider application than the concept of objects, however, applying to the design of a computer at the gate level or the transistor level as well as the many software levels. Within the software levels, the concept of levels of abstraction has matured into agreement with the concept of objects [Dahl72]. Each object has at least two levels of abstraction: the object being modeled and the representation of the object. Both the object being modeled and the representation of the object have abstract definitions that describe their behavior.

Direct system support of the concept of objects is provided by capability systems. Capabilities, which are just object references, form the protection mechanism that provides sophisticated control of access on these systems.

Another major field in computing is the design of data base management systems. An

entity in a data base management system is similar to an object. An *identifier* or *candidate key* is similar to an object reference [Fry76]. Although the concepts in data base management systems are similar to the concept of objects, there are some important differences [Hammer76]. The fact that the recent Conference on Data [Data76] was jointly sponsored by SIGPLAN and SIGMOD indicates that many people are hopeful that the concepts in programming languages and data base management systems can be unified.

1.3 Object References

The concept of objects was developed for Simula 67 to improve the conceptual basis for the list processing used to perform simulations. The object reference was introduced to allow one object to refer to another. Pointers were used for this purpose before the development of object references. A pointer, however, is only an address while an object reference contains both an address and a type code. The type code implicitly specifies the operations that are defined on the object and the representation that is being used for the object. Thus the type code identifies the *data type* of the object. A pointer only specifies a location while an object reference specifies a whole object.

Each object reference has a specific object that it is supposed to refer to. When an object, x, is created, an object reference to x is also created. The object reference to x may be copied freely but it must always refer to the object x. The purpose of the object x is to model an abstraction that is known to the user and has been implemented in the computer. Only the operations that are defined in this abstraction may be performed on x. In order to implement the operations on x, however, it is necessary to manipulate the representation of x. Thus each object must have at least two levels of abstraction: the high level abstraction for which only the operations defined by the user may be performed on x and the low level abstraction, i.e. the representation of x, for which operations are defined that manipulate the representation of x. These two abstractions of an object, the high level abstraction and the low level abstraction, are two different objects that share the same physical representation. They differ in the set of operations that may be performed upon them. A high level operation on x is implemented by taking the reference to x and converting it to a reference to the low level abstraction of x and then performing the necessary low level operations on

the low level abstraction of x. It is possible that this low level abstraction of x will in turn be implemented by lower level objects. Eventually the operations on x will be implemented in terms of manipulating the bits of the physical representation of x. At no time, of course, should any reference to x be used to manipulate the representation of any other object. Finally, when the object x is destroyed, i.e. its storage is freed and reused for other objects, any outstanding references to x should continue to refer to x, which is now a destroyed object. As before, a reference to x should not be used to manipulate the representation of any other object. This is somewhat difficult to achieve in a reusable address space, however, where this problem is known as the dangling reference problem. The dangling reference problem is solved in LISP by never destroying an object until all the references to the object have disappeared. Dangling references are not a problem, however, if the address space they point to is not reused.

Now that we have seen the use that is made of an object reference we can begin considering some basic issues of how an object reference is represented. Abstractly, an object reference consists of both a pointer (address) and a type code, but many implementations of programming languages have been able to use addresses as object references without having any type code in the runtime representation. Such implementations have a speed advantage over the use of runtime types on computers whose hardware supports addresses but does not support type codes. If the hardware supported full object references, however, then the system could provide much better support for the concept of objects without sacrificing speed. Let us now examine the use of object references in more detail.

The representation of an object consists of bits which by themselves have no inherent meaning. The type code in the object reference identifies how the bits in the representation of the object are to be interpreted to achieve the object's abstraction. That is, the type code creates a context in which the bits of the representation of the object are interpreted. It is possible to use this context to reduce the number of bits needed in object references stored within the object. For example, if the object being modeled is a grocery list, its representation might contain two object references. The first object reference would specify the first item on the grocery list, while the second object reference would specify the rest of the grocery list. Since the number of different types of items that might be on a grocery list

is very large, there is no meaningful context that can be used to reduce the number of bits in the first object reference. The rest of the grocery list, however, is either the null object or another grocery list object. Therefore only one bit is needed for the type code in the second object reference.

Simula 67 recognized this possibility and required that every declaration of a variable that holds an object reference specify all the data types (classes) of the objects that the object reference could possibly refer to. If only one data type is specified (strong typing), then no data type need be stored at all. Only the address need be stored. A program's context is not always sufficient, however, to reduce significantly the set of data types of the objects that can be referenced from a particular variable (see example above). EL-1 [Wegbreit74] allows variables to be declared to be of type *any*. Context independent object references that completely specify the data type of the object are used for such variables. When a variable has been declared to reference one of a small set of data types, it is not clear that it is more efficient to use an abbreviated special purpose object reference format than to use a context independent format. Although the abbreviated format is smaller than the context independent format, it is necessary for special purpose code to be written to interpret the special purpose format while the code to interpret the context independent format can be used by the whole system.

The most famous list processing language, LISP, uses a context independent format for object references. Unlike Simula 67, there is no mechanism in LISP for declaring the types of variables [McCarthy65]. By using context independent object references, LISP is able to do very well with a very small number of data types. The main data structure in LISP, the list, is very useful because it can organize data of any type.

If we are to build an object oriented computer, however, object references play a much more important role than allowing one object to refer to another. The CPU must use object references in order to manipulate the representations of objects. Object references are the basis of addressing on an object oriented machine [Fabry74]. The general registers on such a machine hold object references, while address calculation uses the addresses in object references. If the registers of the CPU contain context dependent object references

(addresses without type codes) whose context is provided by the currently executing program, then the CPU cannot help in supporting objects. Since the CPU does not know what objects are being manipulated, all of the support for objects must be provided by software. If, however, the registers of the CPU contain context independent object references (containing full type codes) then the CPU can tell which objects are being manipulated and can provide additional support for objects. Thus the object references stored in the CPU registers should be context independent.

The formats of object references used in current programming languages are not of much help when deciding what format of object reference should be used in an object oriented machine because the low level implementation of a programming language is profoundly affected by the machine it is running on. Most current computers support addresses but do not support any sort of type code with the address. If we are able to specify new hardware to support objects, however, we will not be operating under the same constraints used to develop object references for these subsystems and so should not be surprised if the format of object reference we arrive at is rather different from the format used for any of these subsystems.

If the hardware supports context independent object references, then there will be little advantage to using context dependent object references within objects since extra time would be required for converting a context dependent object reference to the context independent form that is interpreted by the hardware. We should expect few languages on such a system to require strong typing. Instead, languages would encourage the use of the very general data structures that are possible when context independent object references are used.

An object oriented machine must use object references to refer to very small objects such as integers and booleans as well as larger objects. If the abstract definition of an object does not allow side-effects, then it is possible for the entire representation of a very small object to be held in the object reference. Thus small integers and booleans may be referred to with object references that do not contain addresses but contain the actual integer or truth value along with a type code. Bit string objects that are defined to have side-effects and

integers that are too large for the object reference must be referred to with object references that contain addresses.

1.4 Are Object References Unstructured?

In the last few years some people have suggested that the use of pointers or references may be unstructured [Hoare73, Hoare75, Kieburtz76], as is the "goto", and therefore programming languages that do not have these constructs should be encouraged. The pointer in PL/I, however, is just an address while an object reference contains both an address and a type code. Furthermore, the pointer in PL/I is an object, while the object reference is not an object; the object reference is merely an implementation mechanism to enable objects to be manipulated. In fact, addresses or pointers are unstructured mechanisms, while object references are highly structured[1] mechanisms. Since the concept of objects is in basic agreement with the requirements of structured programming, it would be surprising if object references were unstructured. It is not unusual, however, for structured mechanisms to be implemented with unstructured mechanisms. Thus although a procedure call is a structured mechanism, it uses a "goto" as part of its implementation. The major difference between a "goto" and a procedure call is that a procedure call is a meaningful use of a procedure object which in turn models such well-known mathematical objects as functions. A "goto", on the other hand, does not correspond to a meaningful operation on a conceptually meaningful object. Similarly a pointer or address is an unstructured mechanism, while the object reference is highly structured even though an address is an important part of an object reference. An address does not correspond to a meaningful object, while the object reference is the key to manipulating abstract objects.

A major reason why some people have begun to feel that references may be

1. References that conceptually contain a data type and an address have been defended in the literature [Berry76]. Hoare and Kieburtz are not opposed to using object references to implement programming languages, but are opposed to making addresses or pointers available to the programmer. Thus there is general agreement that object references may be used to support objects.

unstructured is that many systems have been built that do not guarantee that a reference will always refer to the same object. In particular, many systems have been built in which dangling references can become a problem, i.e. in which a reference to one object can be used to modify the representation of another object after the storage for the first object has been freed. This is not the fault of the concept of the object reference, however; it is the fault of those systems that do not enforce the proper use of object references. Inherent in the concept of objects is the idea that an object reference should only be used to refer to a single object, thus the references to an object may be used to access storage only in the representation of that object. In addition, the concept of objects does not provide for any method of creating object references other than copying an object reference or when creating a new object. It is therefore improper to allow an arbitrary bit string to be converted into an object reference. These restrictions on the use of object references are inherent in the concept of objects and are thus an important factor that cause object references to be structured while pointers or addresses are unstructured.

1.5 Computer System Design

The concept of objects appears to be a fundamental concept in computation that should influence the design of the computer system. There are two major design criteria in computer system design. First, the computer system should be reliable, i.e. it should always be ready to correctly execute a correct program. Second, the system should perform the computation wanted by the user as quickly and efficiently as possible. These two criteria conflict with each other since reliability often has a cost that decreases efficiency during those times that the system is "up" and available. A reasonable balance between these two design criteria must be achieved. The system must execute programs at high speed, but cannot allow erroneous programs to threaten the integrity of the system.

Designing a CPU that deals with objects must be done very carefully. Different sections of computer science stress different aspects of the concept of objects. In order to serve all parts of the field and thereby all users, all of the aspects of objects should be given meticulous attention. It would be tragic to provide hardware support for objects and then find that some applications cannot use this mechanism but must build a separate software

mechanism for objects. Hardware supported objects will obviously be more efficient that software supported objects on the same system. If the hardware mechanism does not provide all of the flexibility that exists in the concept of objects, however, then a particular application that requires the particular element of flexibility that was not provided may be forced to build a separate software mechanism for objects that does provide that element of flexibility.

The concept of objects requires that a certain relationship be maintained between an object and a reference to the object. If the concept of objects is really a fundamental concept occurring in all applications of computers, then a single implementation of the concept will only be used for all applications if the only restrictions in the mechanism are the restrictions that are inherent in the concept of objects. These restrictions are:

1) an object may be manipulated only through the use of a reference to the object
2) an object reference is created only when its corresponding object is created
3) modifications to the address within an object reference are prohibited; in fact, most other modifications to the object reference are prohibited as well, because most modifications to the object reference would violate other restrictions on the use of object references
4) when an object reference is used to access storage, only the storage containing the representation of the object referred to may be accessed
5) an object reference must always refer to the same object
6) each object is a model of an ideal object known intuitively to the programmer; all of the operations on the object in the computer must be consistent with the ideal object being modeled; this restriction is implemented in three parts:
   a) only the data type definition of an object may convert a high level reference to the object to a low level reference to the object
   b) the definition of an object is determined by the programmer who originally envisioned the object or by someone designated by that programmer
   c) all operations on the object, including machine instructions, must reflect the definition of the object

It is important that these be the only restrictions. In addition, however, the mechanism for object references must have a low cost. Thus, in order to achieve a mechanism that can be

used for all applications the following criteria must also be satisfied:

7) an object reference should not be much bigger than an address

8) an object reference stored in one part of the system should be able to refer to an object in any other part of the system

9) objects may be of any size from one bit (truth values) to millions of words and larger

10) object references may be freely copied

11) it should always be possible to define a new type of object

12) when defining a new object, it should be possible to define how operations that already exist on the system, including machine instructions, will work on this object; thus all operations are representation independent to some degree

Some of these design criteria specify properties of the addressing scheme of the system, some specify flexibility that is needed in the system, and some specify restrictions that should be placed on the use of object references, the use of addresses within object references, and the ability to define new operations on an object. Flexibility must be provided in order to serve all users. A computer system will have achieved the necessary flexibility if all the programming languages based on the concept of objects use the hardware supported object reference to refer to objects within the language.

It was once thought that the restrictions on the use of object references could be achieved by relying upon the good will of the programmer without providing any mechanism for enforcing these restrictions; correct programs would naturally adhere to these restrictions just as a good citizen obeys the law. This is the approach taken by PL/I; it has lead to charges that pointers are unstructured. Software reliability can be improved if the restrictions on the use of object references are enforced in some way. Two different ways of enforcing these restrictions have been proposed. The obvious technique is for the hardware to guarantee that the restrictions are not violated. This is only possible if full context independent object references (containing type codes) are used in the CPU to point to objects and to manipulate the representations of objects. The second approach is to design a compiler that would not generate machine code that violates these restrictions even though a random sequence of machine instructions might violate these restrictions [Palme73, Jones76]. If all the software on the system is written in the high level language translated by this

compiler, then all the software would adhere to these restrictions. This thesis will only consider hardware enforcement; nothing will be assumed about the sophistication of the compiler.

1.6 Contributions of the Thesis

This thesis investigates the possibility of efficiently implementing a new computer system that supports objects in hardware and is a natural successor to several existing systems. The complete description and investigation of a new computer system is beyond the scope of a single thesis, so I have chosen a set of problems in the design of the new system and proposed solutions to these problems. The most unusual aspect of the new system is that it supports object references using a single, very large, paged, linear address space. Although the address space must be reused to achieve efficient paging, the system prevents dangling references from causing problems by not reusing a piece of address space unless there are no dangling references to it. All of the processes and disk files reside within the single address space. In addition, the single address space exists for the life of the system. Garbage collection is an important tool that is used to prevent dangling references and to reclaim storage for inaccessible objects automatically. When all online storage is within the address space, the reliability of the entire system is threatened. This problem is dealt with by providing hardware enforcement of the restrictions on the use of object references. Thus the new system can be considered to be basically a capability system that uses a linear reusable address space rather than unique IDs. The linear paged address space on the new system, however, allows many small objects to be placed on the same page but also breaks large objects into pages. Thus high speed memory can be used efficiently while incurring less disk traffic than on capability systems, which only swap individual objects.

There are several contributions in the thesis that make the new system design feasible. The most important contribution is showing how to perform garbage collection in a very large address space. First the address space is broken into pieces called *areas*; single areas are then garbage collected independently from the rest of the system. Garbage collection of a single area is possible because the system automatically maintains complete lists of all inter-area references. This approach is attractive because I have developed a method for

maintaining the lists of inter-area references automatically without incurring much runtime overhead.

Another contribution is associated with the problem of placing objects into appropriate areas. The programmer places objects into areas to create modules of related information and to increase locality of reference. In case a programmer places an object inappropriately, an automatic mover, developed in this thesis, detects the improper placement and moves the object to a more appropriate area. The automatic mover is implemented by the garbage collector. In addition to correcting inappropriate placement of objects, the automatic mover ensures that cycles that extend over many areas will be reclaimed when they become inaccessible even though the areas involved are never garbage collected together in one garbage collection.

The new system is able to revoke access much better than on other capability systems because both the garbage collector and the lists of inter-area links are powerful revocation mechanisms. It is shown that the new system can provide protection by access control lists as on Multics and can provide revocation that is as sudden and as comprehensive as on Multics. Thus there are two ways of specifying protection on the new system: by access control lists as on Multics and by the use of object references (capabilities). The protection provided by object references is more powerful, however.

A fourth contribution of the thesis is the format of the object reference used on the new system, which is not much larger than an address. The current state of the art in the design of capability systems that use a linear address space (see Chapter 2) requires three other fields in the object reference in addition to the address field. The size of each of these three fields is on the order of the size of an address. These three fields are: a type field, an access control field, and a *size* field. Since the minimum size for an address on the new system is 40 bits, these three fields could cause the object reference to be very expensive. The thesis presents techniques for using as few as 5 bits in the object reference for a *size* field, for reducing the access control field to a single bit with the possibility of using the *size* field for access control information as well as for size information, and for reducing the type field to as few as 9 bits without significantly sacrificing the ability to create new data

types.

An important claim of this thesis is that the proposed system is efficient. This claim is supported by comparing the proposed system with several other classes of computer systems that are described in Chapter 2. Evaluation of the efficency of a computer system, however, is very difficult, so this thesis can only perform a cursory comparison of the new system with other systems. Instead of dealing with total performance of the system, only very low level operations on the different systems are compared. The goal is to show that the proposed system operates about as fast as current systems if the programming styles on current systems are transferred directly to the new system. In addition, it is shown that many operations that are not supported well on current systems are supported well on the new system. If programming styles change on the new system to take advantage of the features that it supports well, better performance can be achieved on the new system than on current systems. High performance is achieved on the new system by providing hardware whose prime design criterion is to provide a very hospitable environment for the support of objects and whose secondary design criterion is speed of computation. The criteria of high quality support of objects and high speed computation are both very important, but the quality of the support for objects predominates somewhat.

These contributions go a long way toward allowing the new system to be as efficient as computer systems that do not enforce the restrictions on the use of object references. The thesis covers the major issues in the design of the new system so that the reader can see that the type of system proposed here has merit and warrants further investigation. Among the important issues that are not discussed in depth are: the machine language, techniques for maintaining reference counts correctly and automatically, and the definition of data types. The issues that will be discussed should make the reader aware of the implications of the addressing scheme in the new system, how areas and inter-area links affect the structure of the system, and the possibilities for and problems of storage management in the new system.

types.

An important claim of this thesis is that the proposed system is efficient. This claim is supported by comparing the proposed system with several other classes of computer systems that are described in Chapter 2. Evaluation of the efficency of a computer system, however, is very difficult, so this thesis can only perform a cursory comparison of the new system with other systems. Instead of dealing with total performance of the system, only very low level operations on the different systems are compared. The goal is to show that the proposed system operates about as fast as current systems if the programming styles on current systems are transferred directly to the new system. In addition, it is shown that many operations that are not supported well on current systems are supported well on the new system. If programming styles change on the new system to take advantage of the features that it supports well, better performance can be achieved on the new system than on current systems. High performance is achieved on the new system by providing hardware whose prime design criterion is to provide a very hospitable environment for the support of objects and whose secondary design criterion is speed of computation. The criteria of high quality support of objects and high speed computation are both very important, but the quality of the support for objects predominates somewhat.

These contributions go a long way toward allowing the new system to be as efficient as computer systems that do not enforce the restrictions on the use of object references. The thesis covers the major issues in the design of the new system so that the reader can see that the type of system proposed here has merit and warrants further investigation. Among the important issues that are not discussed in depth are: the machine language, techniques for maintaining reference counts correctly and automatically, and the definition of data types. The issues that will be discussed should make the reader aware of the implications of the addressing scheme in the new system, how areas and inter-area links affect the structure of the system, and the possibilities for and problems of storage management in the new system.

1.7 Organization of the Thesis

Chapter 2 presents an historical perspective on the design of computer systems that shows how hardware support for the concept of objects has evolved. Selected systems that illustrate this evolution have been considered in Chapter 2. In addition, Chapter 2 presents and describes the other computer systems that will be compared with the new system throughout the thesis. Chapter 3 presents an overview of the new system and many of the mechanisms in it. Chapter 4 discusses garbage collection and shows how individual areas can be garbage collected separately once the system automatically maintains the lists of inter-area links. Chapter 5 presents the mechanisms for maintaining the lists of inter-area links and shows that the overhead for maintaining these lists is incurred only when performing operations that are not supported by hardware on other systems, so there is little runtime overhead for maintaining the lists of inter-area links. Chapter 6 discusses areas and how they interact with file directories and also presents the automatic mover. Chapter 7 presents the details of the new format for object references and discusses the mechanisms for providing protection on the system. Revocation of access is discussed and the technique for controlling access by access control lists is presented and compared to protection on Multics.

Chapter 2
Computer System Design: An Historical Perspective

Once the concept of objects is in hand, it is possible to look back at the computer systems that have been developed in the past and see to what extent each system supported objects. In many cases it is possible to view developments in computer system design as advances in the support of objects. This chapter performs such an analysis, showing the techniques that have been tried for supporting objects, and analyzing how each system has fallen short of the design criteria given in section 1.5. At the end of this chapter, it should be obvious that the design of the system presented in this thesis is a logical successor to all of the systems discussed in this chapter. The new system attempts, however, to overcome some of the deficiencies in these systems. The new system will be compared throughout the thesis to the systems presented in this chapter to show that the new system is efficient.

The hardware in the earliest electronic computer systems consisted of a CPU, high speed memory, much slower but larger and more permanent storage, and several I/O devices [Burks46]. This basic type of computer, sometimes called a von Neumann machine, continues to be used today even in some newly designed systems. The large, slow storage is treated as an I/O device but is also used to load a computation into high speed memory. The real computation is supposed to be executed by the CPU in high speed memory. High speed memory is contained within a linear address space consisting of words of storage. Not only are the temporary results of a computation stored in high speed memory, but so is the program. The program can load and store information in any word of high speed memory. Information in memory is only given meaning when a program manipulates that information. A word of memory can be viewed as holding an integer at one instant, so another number can be added to it; at the next instant it can be viewed as holding an instruction, so the program can transfer control to the location; finally the same word can be viewed as holding an address, so the program can use the address to access another memory location. This structure was created more for its ease of construction in hardware than for its superior computational properties. At the time, however, the ability to view data with different meanings at different times seemed to have important computational properties. After all, it allows the programmer to create instructions and addresses that are the result of

calculation rather than just being constants in the code.

The fundamental nature of objects encouraged machine languages that were somewhat consistent with the concept of objects. The address, however, was the closest construct these machines had to an object reference. On all machines an object may consist of several words; it is a record. Often a constant offset from the beginning of the object is used to access a word of the object. Machines that provide address calculation in almost every instruction and seldom store the result of the calculation in a register, such as the PDP-10, IBM 7094, IBM 360, and H6180, provide limited hardware support to objects and object references because this kind of address calculation is consistent with the address calculation needed with object references.

One of the fundamental requirements of objects is that an object reference must always refer to the same object. As long as an entire computation was contained in high speed memory, this requirement was satisfied well enough. It did not take long, however, for programmers to devise problems that required more storage than existed in high speed memory. When a large, slow memory was available that could be used in a random access manner, the programmer was tempted to use this memory to hold objects that were needed for the computation. These objects could only be accessed when they were actually in high speed memory, however. To achieve an increase in the effective amount of memory available, it was necessary to use some sections of high speed memory for different objects at different times. Addresses elsewhere in memory that pointed to these ephemeral objects were very hard to use because they did not point to the correct objects all the time. To cure this problem, the swapping of objects that were referenced from elsewhere in memory was prevented. Thus the amount of high speed memory that was available for swapping was reduced because many objects that were involved in a computation could not be swapped out of high speed memory. Therefore the usefulness of the slow speed memory was limited. This problem was solved by the development of virtual memory on the Atlas computer [Kilburn62]. Virtual memory provides some hardware support to the idea that an address should always point to the same object as long as the object exists.

2.1 Size of the Address Space

When designing a computer without virtual memory in the early 1960's, the choice of the size of the address space was reasonably straightforward. During this time, high speed memory (core) was very expensive. The computer architect could choose an upper bound on the amount of core for the computer system as the size of the address space. Since the address was a physical core address there was no need for more address space[1].

The development of virtual memory removed the correlation between the size of high speed memory and the size of the address space. A virtual memory system may easily have an address space larger than high speed memory. It is only necessary that the *working set* [Denning68, Denning70] of the computation fit into high speed memory to achieve efficient use of the computer system.

If the size of high speed memory no longer limits the size of the address space, how large should the address space be? It is clear that no one has a sound basis for choosing the size of the address space. Recent virtual memory systems seem to take 2^{16} words as a minimum (PDP-11) and go up to 2^{32} words (Multics, MU-5) [Lindsey71, Ritchie74, Organick72].

2.1.1 Small Address Space Virtual Memory (SAV) Systems

In fact, current virtual memory systems can be divided into two distinct types. First there are the Small Address space Virtual memory (SAV) systems. Although these systems have virtual memory, the size of the address space is often too small to hold all of the programs and data of a single computation. As a result these systems require mechanisms that allow program overlays. Such systems deal with disk files by using I/O operations or by using a small portion of the address space to access a small portion of a file. In the latter

---

1. The development of a relocation register for user jobs did not significantly change this viewpoint but did alleviate somewhat the restriction on the amount of usable core storage placed on the system by the size of the address space.

case, the same piece of address space is then reused for another portion of the file. Most virtual memory systems are in this category because it is easy to convert a computer system designed without virtual memory to an SAV system. It requires little more than providing page mapping hardware and writing the portion of the operating system that does demand paging. Much of the operating system may remain unchanged. There is no need to change user software since it already performs overlays and deals with the disk as an I/O device. Examples of SAV systems are: ITS, TENEX, and UNIX [Ritchie74, Bobrow72, Eastlake69].

The SAV architecture is superior to the architecture without virtual memory because a program written for a system with a large amount of high speed memory will be able to run on a system with less high speed memory. The overlay strategy for a program on an SAV system is dependent upon the size of the address space, not upon the amount of high speed memory on the system, thus greater hardware independence is achieved than without virtual memory.

2.1.2 Large Address Space Virtual Memory (LAV) Systems

The second type of virtual memory system is the Large Address space Virtual memory (LAV) system. In LAV systems, all of the programs and data needed by a single computation fit into the address space. As a result such systems do not provide a mechanism for program overlays. When a program uses a file on such a system the entire file is placed in the address space. Portions of the file are not actually moved into high speed memory until they are used, however. The system does not encourage the user to purge a file from the address space unless it is known that no addresses continue to point into it. The address space is large enough that it is not necessary to reuse the address space for such files. Examples of LAV systems are Multics and MU-5.

LAV systems do a better job than SAV systems of supporting the concept of objects. As long as an object exists within a single computation, it retains the same place within the address space on an LAV system, while SAV systems may violate this requirement by the use of overlays. In addition, LAV systems begin to support the idea that an object anywhere in storage should be manipulated through the use of an object reference.

The main feature of an LAV system is that all of the storage for a single computation fits within the address space. Most computations make use of only a small amount of the storage on a system. If the system programmers of an LAV system are influenced only by users who never have computations that use a large amount of storage, then their LAV system could have an address space that is many times smaller than the total amount of on-line storage on the system. If, on the other hand, there are influential users who occasionally have computations that use a large amount of storage and make use of a significant fraction of the software on the system, then either the address space will be almost large enough to handle all the storage on the system or the system will slowly become an SAV system as mechanisms for performing overlays are written and begin to be used. Thus the VS/2 system for the IBM 370, which only has 2^{24} bytes in the address space but has a structure that is very similar to the structure of Multics, may in fact be used as an SAV system because its address space is so small. If a system has an address space that is so large that it can be shown that it can contain and use all of the on-line storage on the system, then I call that address space a very large address space. Both Multics and MU-5 have address spaces that are so large that they are almost very large address spaces.

2.2 Single Address Space

LAV systems supply a different address space to each process or computation. Thus the object references on an LAV system are only valid within the context of one process. Although it is possible for an object reference on an LAV system to point to any word of storage on the system, it is not possible for an object that exists within permanent storage (within a file) to contain object references to other objects in the system[1]. Similarly it is not possible for object references to be communicated from one process to another. Any address

---

1. Computer systems often provide file systems that implement a kind of single "address space" that maps character strings into files. The file system is a large body of software that implements this "address space". A basic premise of this thesis is that any mechanism for manipulating objects should use an address space that is interpreted directly by hardware. The "address space" provided by a file system is thus inadequate for use within object references.

stored in a permanent object would point to different parts of the system in the different processes that used the object. This violates the most basic requirement of object references. If a single address space were provided for all the processes for the entire life of the system, then an address stored in a permanent object would have the same meaning for all processes. An address would always point to the same object.

Unlike an LAV system, however, the size of the address space on a single address space system places an upper bound on the amount of online storage that can realistically be used, so a single address space system must have a very large address space. If the address is too big, however, then several bits in each address will always be zero and the hardware needed to interpret these bits and the storage used for these bits will be wasted. Thus some attempt must be made to keep the address space as small as possible. Multics wastes address space when it uses an entire segment for a short file. If such waste is eliminated, then the size of an address on a single address space system may be only a few bits larger than the size of an address on Multics.

An LAV system needs a separate memory map for each process, but a single address space system needs only one virtual memory map. The memory map on a single address space system is only modified as pages are moved in and out of high speed memory. The memory map is common to all processes, so the operation of switching processes is faster than on LAV systems because it is not necessary to switch the memory map.

The concept of segmentation which is so prevalent on LAV systems such as Multics is much less important on single address space systems. An important purpose of segmentation is to allow a section of a memory map covering many pages (one segment) to be independent of the address space it is used in. On such a system an address space is defined in a two step process (see Figure 1). First there is a table of segment descriptors. Each segment descriptor points to the page table for that segment. The position of a segment descriptor in the segment descriptor table specifies the segment number for that segment. If a segment is used in several address spaces whose segment descriptor tables are all in high speed memory, only one copy of the page table for the segment is needed. Each address space, however, may have a different segment number for that segment. Thus, segments play an important role

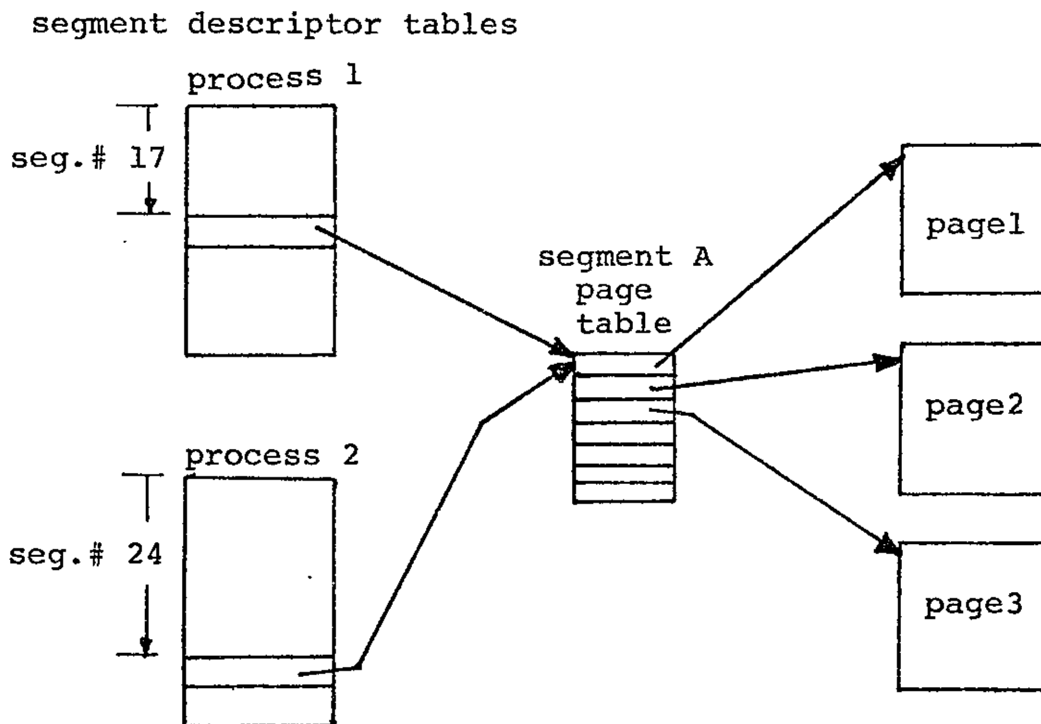


Fig. 1. Segmentation reduces the memory needed for mapping multiple address spaces.

in reducing the overhead for creating and maintaining multiple large address spaces: a task that is unnecessary on a single address space system.

A problem with a single, large address space is that if a program contains an error, it can cause any process executing that program to go marauding through the address space. Even on an ideal system there is no way to protect a computation that exercises a bug from itself, but other computations, files, and data bases that are unrelated to the erroneous computation should not be affected by it. Both SAV and LAV systems use multiple address spaces to limit the damage that can be caused by a bug.

2.3 Capability Systems

Capability systems, on the other hand, provide very sophisticated protection while operating in a single address space for the entire system [Fabry74]. Capability systems provide their protection by enforcing the restrictions, inherent in the concept of objects, on

the use of object references. A capability is a context independent object reference, containing type information as well as an address. If a program has access to a capability, the program may perform an operation on the object pointed to by the capability. The access control field in a capability specifies which operations may be performed on the object. Each object can be viewed in at least two different ways, however: as a high level object on which only the operations that are meaningful for the object being modelled are allowed, and as a low level object on which operations are defined that allow the representation of the object to be manipulated. Since the purpose of an object is to implement its high level abstraction, the users of an object should only be able to perform the high level operations on the object. The only programs that can perform a low level operation on the object are those programs that define the data type of the object. Thus sophisticated protection concentrates on limiting the set of high level operations that may be performed on the high level abstraction of an object. There is not much point in limiting the representation dependent operations that may be performed by the data type definition on the low level abstraction of an object because the data type definition is highly privileged code: it implements the high level abstraction of the object, so it must work correctly. Limitations should be placed on the least trusted users of an object, not the most trusted. Many capability systems, however, have put much effort into limiting the operations on the object that the data type definition can perform as well as or sometimes instead of limiting the high level operations that can be performed on an object. Thus the classic read, write, and execute bits for controlling access limit the low level operations rather than the high level operations on an object.

The restrictions on the use of object references (capabilities) enforced by capability systems are shown in Table 1. These restrictions form two sets. The first set of restrictions, the low level restrictions, has two purposes. First, it ensures that the low level operations on an object (the representation dependent operations) can only manipulate the representation of the object referred to by the object reference. Second, the low level restrictions ensure that all references to an object were obtained by making copies of other references to the object which eventually were copied from the reference to the object that was created when the object itself was created. These two basic properties allow the distribution of references to the object (and thereby access to the object) to be controlled by the software that created the

Table 1. Restrictions on the Use of Object References

Low Level Restrictions

1) an object may be manipulated only through the use of a reference to the object
2) an object reference is created only when its corresponding object is created
3) object references may be copied, but most modifications to an object reference are prohibited, especially modifications to an address within an object reference; thus the system must be aware of the location of all the object references on the system
4) when an object reference is used to access storage, only the storage containing the representation of the object referred to may be accessed
5) an object reference must always refer to the same object

High Level Restrictions

1) each object is a model of an ideal object known intuitively to the programmer; all of the operations on the object in the computer must be consistent with the ideal object being modeled; this restriction is not directly computable: it must be left to the judgment of the programmer; this restriction is implemented in three parts:
   a) only the data type definition of an object may convert a high level reference to the object to a low level reference to the object
   b) the definition of an object is determined by the programmer who originally defined the object or by someone designated by that programmer
   c) all operations on the object, including machine instructions, must reflect the definition of the object

---

object. The low level restrictions on the use of object references also ensure that the representation of an object can only be manipulated by use of a reference to that object. The low level restrictions do not provide for limiting the operations on an object, however. This is left to the high level restrictions.

Each object is supposed to model an ideal object. The high level restrictions allow operations that are not defined on the ideal object to be prohibited. Although the most important use of the high level restrictions is to prohibit low level operations on the high level abstraction of an object and to allow the programmer to limit the high level operations on an object that may be used by one of the users of an object, the high level restrictions merely state that the programmer may limit the operations that may be performed on an object. If the programmer decides to prohibit some low level operations while allowing others, he may do so by making use of the high level restrictions on the use of object references.

The high level restrictions are implemented by associating with each object a data type definition that specifies which operations may be performed on objects of that type and how these operations are to be performed. These operations may be defined to check the access control information in the object reference or in the representation of the object and only operate if these checks succeed. Only a data type definition for type D is allowed to convert a high level object of type D to a low level object. This conversion is performed by modifying the access control field in the object reference. Thus we may talk about a high level reference to an object of type D and a low level reference to the object. Since the data type definition must implement interpretation of the access control field in an object reference, the data type definition can be trusted to restrict distribution of its low level reference to the object. The code that creates an object of type D and is given the first low level reference to the object is considered to be part of the data type definition of type D. Thus the data type definition for type D is given the first low level reference to objects of type D and only the data type definition may convert high level references to objects of type D to low level references. Consequently, the data type definition can control the distribution of low level references to objects of type D. If the data type definition does not distribute any low level references to objects of type D to programs outside of the data type definition of type D, then only the data type definition can manipulate the representations of objects of type D. The high level restrictions on the use of object references allow access to an object to be arbitrarily controlled by the data type definition. As with other objects, the ability to modify a data type definition is controlled by the software or the user who created the data type definition.

It is surprising how difficult it is to enforce the low level restrictions efficiently without infringing on the flexibility of object references listed in section 1.5. It is also difficult to enforce the high level restrictions efficiently without limiting flexibility, but it is possible to separate the implementations of the low level and the high level restrictions. Once it is recognized that the type code in the object reference should point to the data type definition for the object, then the hardware can be made to map the type code into a pointer to the data type definition. Implementation of the high level restrictions is then largely a matter of specifying the representation of the data type definition and the techniques for converting a high level reference to a low level reference. This thesis deals primarily with implementation

of the low level restrictions and conversion of a high level reference to a low level reference.

Although sophisticated protection of information can be achieved by enforcing the high level restrictions as well as the low level restrictions, a great deal of system reliability is gained even when only the low level restrictions are enforced. Thus if all the object references on a system were low level object references then an erroneous program could still only damage the objects for which it had references. Operations such as accessing outside of the bounds of an array or using a character string as an address would cause errors even on such a system, thus halting the erroneous computation. Although it would be theoretically possible for an erroneous program to carefully follow long chains of object references and to modify objects that were remote from the error, such as critical system data bases or the computations of other users, in practice erroneous programs, unlike malicious programs, are not written carefully enough to do this without violating one of the low level restrictions on the use of object references. Thus a certain amount of system reliability can be gained by enforcing the low level restrictions on the use of object references without enforcing the high level restrictions. The amount of reliability thus gained is at least comparable to the reliability achieved by multiple address spaces on SAV systems. Thus enforcement of the low level restrictions on the use of object references regains the reliability that was lost by deciding to place all the users and all the files into a single very large address space.

Once the low level restrictions on the use of object references are enforced, however, it is foolish not to enforce the high level restrictions as well since they provide such sophisticated protection and also improve reliability further by increasing the number of illegal operations on objects and by decreasing the number of objects that can be accessed by the remaining legal operations. Furthermore, the cost of enforcing the high level restrictions can be avoided for those objects for which it is not necessary merely by distributing low level references for the objects instead of distributing high level references for the objects. Thus the costs for sophisticated protection can be limited to those objects for which sophisticated protection is necessary. Therefore the costs for enforcing the high level restrictions on the use of object references should be attributed to providing sophisticated protection, while the costs for enforcing the low level restrictions should be attributed to maintaining system reliability.

Capability systems were developed to provide protection in computer systems. They were not designed to support simulation or list processing. As a result, capability systems have concentrated on enforcing the restrictions on the use of object references and have sacrificed the flexibility and low cost that object references need to have to be used for other applications as well. SAV and LAV systems, on the other hand, are more concerned with efficiently implementing programming languages and so have concentrated more on providing flexible and low cost object references but have not been concerned with enforcing the restrictions on the use of object references.

A capability system provides protection by placing restrictions on the use of object references (capabilities) that are not enforced on non-capability systems. It may be necessary for a capability system to have a fair amount of specialized addressing hardware to enforce these restrictions while still making it easy to use object references (capabilities). In order to justify specialized hardware it is necessary for it to be used very frequently. Hardware to support capabilities would be guaranteed of heavy use if programming languages such as LISP, APL, and BASIC would use capabilities for the object references needed in these languages. The speed advantage achieved by using a hardware supported mechanism rather than a software implemented mechanism should cause languages such as PL/I, Algol 68, and Simula 67 to use object references (capabilities) for pointers and reference variables, especially since the low level restrictions on the use of object references are consistent with the defined use of pointers and reference variables in these languages[1]. The more object references (capabilities) are used, the better will be the protection provided by them and the less painful it will be to specify needed protection. Unfortunately, no capability system has

---

1. Some people do not realize that although pointers can be abused in PL/I, these abuses are usually illegal PL/I. So many implementations of PL/I have been made that do not enforce these restrictions that some people think that PL/I allows these abuses. Quoting from the Multics PL/I Language manual:

"All of these mechanisms require that the variables that share a generation of storage have identical data types and alignment <attribute>s." [4-19]

". . . it is an error to take the "addr" of a parameter and assign the resulting locator value to static storage and subsequently, in another block activation, use the locator value." [4-21] [Honeywell72]

yet been created on which it is practical to implement LISP by using capabilities for the object references in LISP. It is clear, however, that some people have been hoping that this would become possible [Fabry71]. A major goal of the current thesis is to propose a new way of constructing capability systems that will allow capabilities to be used for object references in all programming languages.

2.3.1 Direct Capability (DC) Systems

We may now face the issue of how capabilities should be implemented. A CPU, in the last analysis, must perform every memory access with a physical address. It is therefore tempting to place a physical address in the capability. A capability system on which capabilities contain the physical address of the objects they reference will be called a Direct Capability (DC) system. The B6700 [Organick73] and the Rice-2 [Feustel72, Feustel73] are DC systems. The B6700 and the Rice-2 are not usually considered to be capability systems, but they do use special hardware to enforce the low level restrictions on the use of object references. The object references supported by these systems are actually used by programming languages such as ALGOL 60. For the purposes of this discussion these systems illustrate the difficulties with DC systems.

Although the use of physical addresses within capabilities sounds very efficient, it does not handle movement of objects between high speed memory and disk very well. Whenever an object is moved in physical memory, all of the capabilities for that object must be modified. Since the system performs this modification, however, it does not violate any of the restrictions on the use of object references because the object references for the object moved continue to point to the same object. The modifications of the object references are invisible to the user and merely ensure that the proper relationship between the object and its references is maintained even though the location of the object has changed. Actually, only the capabilities in high speed memory need contain the addresses of the objects in high speed memory, while capabilities stored on disk would always contain the disk address of the object. When an object is brought into high speed memory, all of the capabilities in high speed memory for the object would be modified to point to the location of the object in high speed memory. In addition, all the capabilities stored within the object would be

modified to point to the proper object in high speed memory. Even the problem of finding and modifying only the capabilities that are in high speed memory is a very difficult job, however, especially if the programmer is able to copy capabilities easily [Organick73, Fabry74]. Usually DC systems solve this problem by preventing capabilities from being copied or by encouraging programming styles in which capabilities are not copied. Either of these solutions prevents languages such as LISP and Simula 67 from using capabilities as object references within these languages. DC systems are likely to provide means of using a capability "indirectly" as with the Stuffed Indirect Reference Word (IRWS) on the B6700 [Organick73, pp. 98-99]. An IRWS can be used in place of the capability it points to in order to alleviate the requirement that capabilities not be copied, but this solution forfeits the efficiency of using physical addresses directly. At least one extra memory reference is required on each access: the same overhead required for virtual memory. In fact, the use of physical memory addresses in object references is not more efficient than the use of virtual addresses. "Virtual" address spaces have the potential for being designed specifically to support the concept of objects.

2.3.2 CUID Systems

The problem with using a physical address in a capability arises when the physical location of the object changes. This problem can be solved by defining a virtual address space in which the object's location never changes. All of the address spaces discussed so far have contained words of storage. This is not necessary for the "address" space used by a capability. It is only necessary to provide a separate ID number or "address" for each object [Dennis66]. A capability system on which capabilities contain a unique ID number that identifies the object forever will be called Capability systems with Unique IDs (CUID systems). CAL-TSS, developed at Berkeley [Lampson76], the Typical Capability System

described by Redell [Redell74], and HYDRA [Wulf74] are CUID systems[1].

Unique IDs are never re-used and the ID number associated with an object never changes. Creating a reference to a new object is a very simple operation. The operating system maintains a counter, N. All IDs less than or equal to N have already been allocated to objects, while no IDs greater than N have been allocated. To create a new object, N is incremented by one and its new value is used as the ID of the new object. The size of the ID space is made large enough so that it will never be exhausted. Usually a size of 2^{64} IDs is chosen (1.8×10^{19}). This is the number of microseconds in 600,000 years. The size of the ID space is chosen by selecting a life-time for the ID space and a maximum rate of new object creation. This immediately gives an ID space size [Redell74, p. 28]. Since object IDs are never changed, there is no need to change the IDs or "addresses" in capabilities and therefore there is no need to place restrictions on the copying of capabilities.

Although it is not possible to exhaust the ID space it is very easy to exhaust the storage on the system. The operating system maintains a catalog of where each object is in physical memory. In order to move an object in physical memory, this catalog is modified in just one place. Similarly, when the programmer wishes to change the amount of physical storage associated with the object, the object is merely moved in physical memory and the catalog is modified in one place. It is therefore very easy to implement dynamic arrays [Dennis65] on a CUID system. When an object is no longer needed the storage associated with it is reduced to zero and the entry for the object is completely removed from the catalog. The catalog therefore maps a sparse ID space onto physical storage.

A serious problem for every subsystem on systems with more conventional word addresses is storage management. The storage management problem that has received the

---

1. Fabry has advocated CUID systems [Fabry74]. He did not say much about HYDRA or CAL-TSS because the hardware on these systems did not support capabilities: only the operating systems did. Fabry points out that the Plessey 250 (which is halfway between a DC and a CUID system) did a reasonably good job of providing inexpensive capabilities that could be stored in arbitrary data structures, but he then cited a few inefficiencies in the Plessey 250 and suggested that CUID systems should be built instead.

most attention is storage fragmentation. A strength of CUID systems is that they are able to handle storage fragmentation very well. "Storage" fragmentation is not a problem in the ID space since this "address" space is never reused. A CUID system must map each unique ID into a physical address space, however. The physical address space is basically a linear address space consisting of words of storage. The physical address space must be reused. Storage fragmentation is a serious problem only when the assumption is made that an object cannot be moved. The strength of CUID systems, however, is that the physical locations of objects can be changed easily to implement automatic swapping. Thus if an object fragments some available storage, the object can be moved to combine the two blocks of available (free) storage adjacent to it (see Figure 2). Indirection through a central catalog in a CUID system is such a powerful tool for handling storage fragmentation that it has been suggested that ALGOL 68 implement references using this technique in order to deal with storage fragmentation [Baecker70].

2.3.3 Garbage Collection

Storage fragmentation is not the only storage management problem, however. Since capability systems do not allow any operation on an object to be performed without a

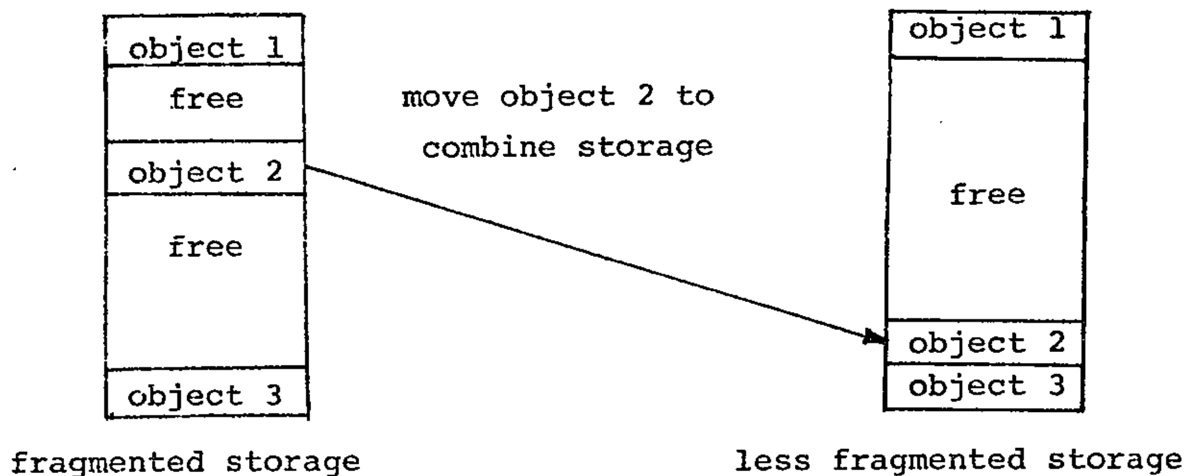


Fig. 2.

reference to the object, and since object references must be copied rather than being built from arbitrary bit strings, it is possible for an object to become inaccessible if all of the references to the object have either been destroyed or stored in other inaccessible objects. If no program on the system can obtain a reference to the object then the object is inaccessible. One of the operations on an object on most capability systems is deleting the storage used by the object. If an object becomes inaccessible, however, it is no longer possible to perform any operation on it, including deleting the storage associated with the object. Thus when an object becomes inaccessible it is impossible to use the storage associated with it since no operation can be performed on the object, but it is also impossible to delete the storage from the object. Thus the physical storage used by inaccessible objects is wasted. There are two ways of dealing with this problem. It could be thrust upon a higher level of software to ensure that no object ever becomes inaccessible. A file system takes this approach when it maintains names for all the objects on the system. Thus if an object becomes inaccessible except to the file system, the user can supply the name of the object to the file system and obtain a reference to the object. Usually the file system deletes the storage for an object when the name for the object is deleted from the file system. This approach is only practical if objects are large enough so that their existence can be largely determined explicitly by the user. The other approach is for the system to determine automatically when objects become inaccessible and reclaim the storage for such objects.

The problem of deciding when to reclaim the storage for objects has been a major difficulty in programming languages. The storage for an object should be reclaimed when the object is no longer needed. The problem is that one object may be used for many purposes, thus making it very difficult for any one program to determine when the object is no longer needed. The concept of objects requires that a program have a reference to an object before the program can use the object. It is possible for a utility program, called the garbage collector, to determine which objects have become inaccessible and reclaim the storage for these objects. Once such a utility program exists, however, we can ask whether it is necessary to reclaim storage in any other way. As long as programs do not keep accessible object references that will never be used, the garbage collector will reclaim the storage for all objects that are no longer needed. This programming convention works very well in practice and is usually followed by the programmer without effort. The use of a garbage collector

makes the job of programming easier because the programmer need not try to determine when he is finished with an object; the garbage collector does this automatically.

LISP is a programming language that automatically reclaims storage for inaccessible objects and does not allow the programmer to free objects explicitly. LISP has been used for over 15 years for artificial intelligence applications that have sophisticated storage management requirements. Automatically reclaiming the storage for inaccessible objects has thus been shown to reclaim storage adequately for many demanding applications. The ease of programming in LISP due to automatic reclamation of storage has been an important factor in the frequent choice of LISP for symbol manipulation problems.

The concept of objects requires that if an object is to be used in the future, a reference to the object must be retained. If an object becomes inaccessible, however, this implies that the object will never be used again. Thus the concept of objects suggests that the storage for inaccessible objects should be reclaimed. HYDRA tries to implement this requirement in two ways. A count is maintained of all outstanding references to an object. When this reference count reaches zero, the object can no longer be accessed. It has long been realized, however, that reference counts do not reclaim all inaccessible objects. If objects B, C, and D exist on the system as shown in Figure 3, these objects are said to form a *cycle*. If there are no other

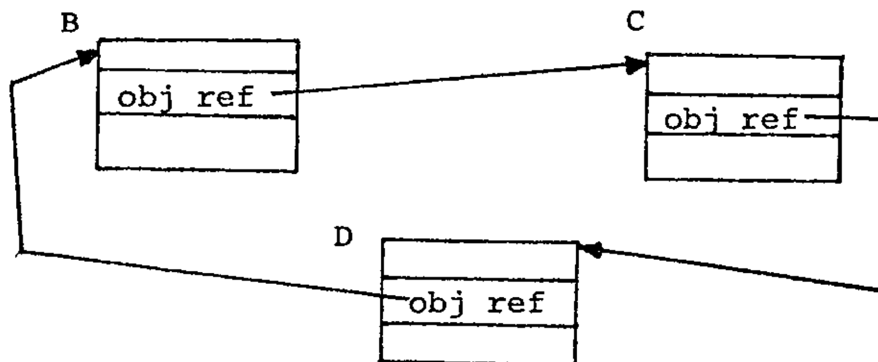


Fig. 3.

references to these objects than those shown in Figure 3, then the reference count for each object is non-zero but none of the objects are accessible from the rest of the system. Reference counts do not recognize such inaccessible objects.

The only known way to recognize all inaccessible objects is to find all of the objects that are accessible and then free all other objects. This must be done by first finding the immediately accessible objects. All objects referenced from accessible objects are also accessible. The process of finding all the accessible objects and reclaiming the storage for the inaccessible objects is known as garbage collection. Experience with garbage collection in LISP has shown that it is a method of reclaiming storage that can run quickly and can prevent the gradual erosion of storage into useless data structures.

In LISP, only the temporary storage for a single computation is garbage collected. On a capability system, however, all the accessible objects on the whole system would have to be found. This rapidly becomes inefficient as the total amount of online storage increases. By the time the total of online storage is 10^{12} bits, this technique is totally impractical. When HYDRA had only 30 million bytes of storage, garbage collection required about three minutes and was only done once a day. The size of the ID space on HYDRA does not place any restriction on the total amount of storage that can be used on the system. The fact that the garbage collector currently places a restriction on the amount of storage that can be used on HYDRA is a problem that is being studied.

Usually when garbage collection is discussed in the context of operating systems, its use is limited to handling the problem of fragmentation of storage. Thus files on disk may be garbage collected, or the jobs in high speed memory may be garbage collected. With this view, entire files or entire jobs are single objects. The problem being solved in these cases is that the available memory has been broken into many pieces that are too small to use for new files or jobs. The garbage collection moves all of the existing objects in storage so that the available storage is in one large block. Usually operating systems do not have the ability to delete the storage for objects, only the user has this ability, so the garbage collector does not try to decide what objects should be deleted. If the system does not enforce the proper use of object references, then the system has no way of deciding which objects a program

will no longer use. A program on such a system can compute an address out of any miscellaneous piece of information and so its entire address space is accessible. The system must rely on the user to tell it when a piece of storage is no longer needed. If, however, the proper use of object references is enforced, as on a capability system, then the system can find all the accessible object references and can be sure that only one object will be used by each reference, so it becomes possible for garbage collection on a capability system to delete the storage for inaccessible objects automatically as well as to handle storage fragmentation. CUID systems do not need to do garbage collection to solve storage fragmentation problems. A CUID system need only modify the ID map to move an object within storage. Although this tool can solve storage fragmentation problems, it does not eliminate the need for garbage collection. The opportunity to reclaim storage for inaccessible objects automatically with a single mechanism that would be used by all subsystems is too valuable to pass up.

2.3.4 Small Objects

Although current CUID systems use a garbage collection algorithm that is not practical for large amounts of online storage, this is not the major problem with CUID systems. The major factor that discourages programming languages from using capabilities on CUID systems for object references within the languages is that CUID systems have too much overhead per object to make the use of small objects efficient.

CAL-TSS and HYDRA had a great deal of overhead per object because capabilities were supported only by operating system software on these systems: there was no hardware support for capabilities. I have already mentioned that capability systems must have hardware support to be efficient, so I will ignore this aspect of CAL-TSS and HYDRA. I am concerned about sources of overhead that would still remain if special hardware were designed specifically for capability systems [Redell74 & Fabry74].

The major factor that makes small objects inefficient in current capability systems is the fact that they swap objects individually between high speed memory and disk. The overhead involved in a disk operation is extremely high for objects containing only two or three words. The average size of objects may be as small as six words each. A recent study of the average size of objects in Algol 60 programs [Batson77] found that the average size

of an activation record is 13-18 words while the median size of an activation record is 6-9 words. The study found larger sizes for arrays. Algol 60 does not allow object references to be used within objects to support list processing, however. Whenever an application requires list processing, it is coded in Algol 60 by using an array as an address space and using an index into the array as an object reference, thus causing large arrays to be used in place of many small objects. An object corresponds rather well to a structure declaration in PL/I. Structure declarations consisting of named fields are somewhat similar to the activation records studied by Batson. Batson argues that the median size of arrays is a better estimate of the average size of arrays than the value he measured. Batson also shows, however, that the size of arrays has very little effect on the average size of activation records and arrays combined, so we must consider the size of activation records to discover the average size of objects. Noting that LISP uses two word objects most of the time, I think that the median size of activation records measured by Batson (6-9 words) is a better estimate of the average size of objects than is the average size of activation records measured by Batson. Thus object oriented systems should be designed so that if the average size of objects turns out to be between 6 and 9 words, the efficiency of the system will not be seriously affected. Regardless of what the average size of objects is, however, it has been noted that there are many very small objects. The only way to reduce the per object overhead for disk operations for very small objects is to swap several objects during each disk operation, as has been suggested by M. O'Halloran [Fabry74]. This approach is an improvement only if several of the objects actually swapped during a disk operation are needed in high speed memory. Devising schemes that swap several objects that are all needed at the same time is a currently active area of research.

Chapter 3
The New System - ORSLA

The goal of this thesis is to propose a new way of building capability systems that would enable very small objects to be used extensively. On such a system capabilities would be used by programming languages to reference objects within the languages. A crucial step in this process is to provide garbage collection to reclaim the storage for inaccessible objects. I assume that such a system would make such widespread use of capabilities that a new CPU would be designed for the system to provide extensive hardware support for capabilities.

The user of such a system would be much more aware of using capabilities to reference objects in computation than using capabilities to implement protection. Wulf says that "Hydra supports *references to objects* (called *capabilities*)" [Wulf74, p. 341]. The term "capability" is only appropriate when discussing protection while the term "object reference" is appropriate in any context. The computer system proposed in this thesis is therefore called ORSLA (Object References in a Single, Large Address space). Throughout the rest of this thesis the term "object reference" will be used instead of the term "capability". The name, ORSLA, emphasizes the fact that the object references will be used to refer to objects within programming languages but it should never be forgotten that object references will provide protection as well. The use of object references for protection will have far-reaching effects on the implementation of the system.

3.1 Address Space

The first problem in the design of ORSLA is: what kind of address space or name space should be used to implement object references? Direct capability systems used physical memory addresses and failed to meet all the criteria given in section 1.5 for the support of object references. Some sort of virtual address space or name space should be used. Capability systems with unique IDs also failed to meet all the criteria in section 1.5, however. ORSLA investigates the possibility of using a linear, paged address space reminiscent of virtual memories in non-capability systems.

There is only one address space on ORSLA. The same address space is used for all users and contains all of the files on the system as well. The size of the address space therefore places an upper bound on the total amount of on-line storage that ORSLA can make full use of. The upper bound on the amount of online storage on ORSLA should not be smaller than 10^{12} bits. There are several factors that lead to this conclusion. First, memory technology is advancing rapidly to supply faster, cheaper memory, so a computer system design should incude a safety factor to prevent the system from becoming obsolete shortly after it is produced. Second, ORSLA will not be placed in a single installation; it will be a complete computer system that will be sold to many people; the upper bound on the storage should be large enough to satisfy all of the ORSLA installations. Third, it is unrealistic to expect a user to be able to switch from an ORSLA system using one size of address space to another ORSLA system using another size of address space. Probably all of the software on the system will be dependent upon the size of an address. Although this dependence is not desirable, and although using objects and high level languages helps reduce this dependence, these tools are not presently powerful enough to ensure that users can be moved to a system with a different size of address space. Thus the upper bound on the online storage on ORSLA should be chosen so it will be acceptable for all ORSLA installations. 10^{12} bits may be too small, however. After all, 10^{12} bit memories are already commercially available. I think that 10^{15} bits of storage is a good maximum on the amount of storage that can effectively be used by a system containing about 10 processors in the speed range of one million instructions per second. Assuming 100 bit words, it would require more than 3 years of operation to access 10^{15} bits of storage at the rate of 10^6 words per second. 10^{15} bits of storage has been estimated to be the amount of information in text contained in the Library of Congress [Licklider65]. Although 10^{15} bits of storage may not be considered to be sufficient as an upper bound on all of the write-once storage on a computer system, as an upper bound on erasable storage, it is beyond the comprehension of present-day programmers. Thus the upper bound on the amount of online storage on ORSLA should be between 10^{12} and 10^{15} bits. Assuming 64 bit words, 10^{12} - 10^{15} bits of storage are about equal to 2^{34} - 2^{44} words of storage. In order to ensure full use of this much physical storage it is necessary to have an address space of 2^{40} - 2^{50} words (see appendix A for more details). This means that most pages of virtual address space do not have any physical storage assigned to them. This will turn out to be a useful feature for the implementation of certain

large objects. Thus an address on ORSLA will use between 40 and 50 bits. At the time ORSLA is actually built, one of these sizes will have to be chosen.

There are several factors that influenced the decision to use a linear paged address space on ORSLA rather than a unique ID space. There are two properties of ID spaces that could cause a designer to choose the ID space. Experience has shown that these properties do not lead to the significant increase in efficiency that might be expected, however. The first positive property of a unique ID space is that since IDs are never reused, the dangling reference problem is solved without needing a garbage collector. The dangling reference problem must be solved, of course, if object references are to be the basis for protection on the system. An ID space also allows storage fragmentation to be solved by the operating system without using a garbage collector. In a reusable address space, on the other hand, garbage collection is necessary to solve the dangling reference problem, i.e. an address can only be reused if no object references exist for the object that used the address previously. A linear paged address space must reuse addresses to maintain adequate locality of reference and so must have garbage collection. Experience with CUID systems shows, however, that garbage collection remains necessary to reclaim the storage for inaccessible objects. Thus garbage collection was provided on HYDRA even though it was known that garbage collection would present efficiency problems. This thesis shows how to make garbage collection on ORSLA practical. Once a garbage collector is available, it becomes more efficient to have a reusable address space rather than a nonreusable address space because the reusable address space is smaller and allows the use of smaller addresses.

The second positive property of an ID space is that it can be smaller than a linear paged address space. Each ID specifies an object and each address specifies a word, but the average size of objects is larger than one word. As we saw above, a linear paged address space must be reused, so this second property only results in an ID space that is smaller than an alternative linear paged address space if the ID space is also reused. At one time, when objects were thought to correspond to files, it was thought that objects were quite large. As it becomes clear that objects correspond to objects within programming languages, however, estimates of the average size of objects have decreased. Since the average size of objects in Algol 60 programs is between 6 and 9 words (see the end of Chapter 2), the size of an ID

space cannot be much smaller than the size of an alternative linear paged address space.

Thus the advantages of an ID space are not as great as was once hoped. Furthermore, there are several disadvantages to an ID space. An ID space is much more difficult to map onto physical memory than a linear paged address space because each ID must be mapped individually, while an entire page can be mapped as a unit. Thus an entry is needed in the ID space map for each object, which may have an average size of between 6 and 9 words, causing the ID space map to use between 10% and 25% of the amount of storage used for the objects themselves. Since typical page sizes are between 250 and 1000 words, the virtual memory map for a linear paged address space uses less than 1% of the amount of storage used for the objects. The extra difficulty of mapping an ID space is not only reflected in a very large ID map. If we assume that virtual memory mapping or ID space mapping occurs on every memory access then ID space mapping may be slower than mapping a linear paged address space. Each memory access must map the ID or virtual address to a physical address and then access the physical location in high speed memory. The ID map or page map for all objects or pages in high speed memory is kept in an auxiliary pseudo-associative memory whose access time is at least as fast as high speed memory. The effective access time is the access time of the auxiliary associative memory plus the access time of high speed memory. If the auxiliary memory holds mapping information for all of the objects in high speed memory, then the size of the auxiliary memory for a linear paged address space is less than 1% of the size of high speed memory, while the size of the auxiliary memory for an ID space is at least 10% and maybe as much as 25% of the size of high speed memory. Economics determines the ratio of the speeds of the auxiliary memory to high speed memory. The smaller the auxiliary memory, the more may be spent per bit on the auxiliary memory to yield a faster memory. Thus if the speed of high speed memory is held constant, the effective access time on a system with a linear paged address space will be less than the effective access time on a system with an ID space because the small auxiliary memory for the linear paged address space will be faster than the large auxiliary memory for the ID space.

It is often assumed that the rest of the details of mapping an ID space are as complicated as mapping a linear paged address space, but in fact mapping the ID space is

more complicated and may result in limitations on the size of objects in the ID space. A memory access on an object oriented computer begins with an object reference and an offset within the object that identifies the word that is to be accessed. In a linear paged address space, the offset is added to the address within the object reference to find the virtual address of the word being accessed. If pages are 2^p words long, this address is split into two parts: the high order bits specifying the virtual page number and the low order p bits specifying the offset within the page. The auxiliary associative memory is accessed with the virtual page number to find the physical location of the page in high speed memory. The offset within the page is then added to the physical location of the page to obtain the physical address of the location being accessed.

In an ID space, on the other hand, the ID in the object reference is concatenated with the offset as shown in Figure 4. The high order bits of this large address are used to access the auxiliary associative memory to find the physical location of the appropriate page of the object. The offset within the page is then added to this physical location to find the physical location of the word being accessed. Usually ID space systems assign only two words of high speed memory to a two word object rather than requiring an entire page of high speed memory to be used for such a small object, so the auxiliary memory also contains the size of the page which is checked against the offset within the page before accessing the location. If objects are not paged, i.e. $m = n$, then adding the offset to the physical location of the object assumes that the entire object is in high speed memory, but an object can only be placed

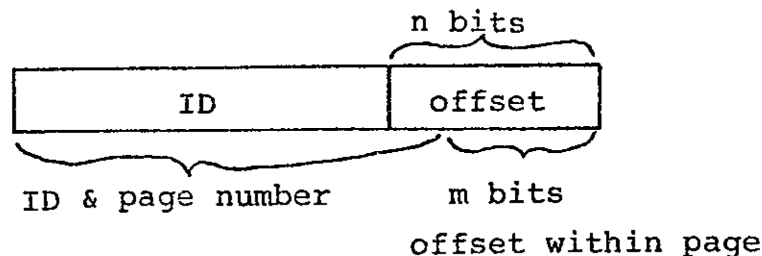


Fig. 4. Address Mapping in an ID Space

entirely into high speed memory if it is smaller than high speed memory. If objects are not broken into pages then the largest object that can be used on the system is a little smaller than the size of high speed memory; this size is very implementation dependent, however. Breaking large objects into pages reduces this restriction and increases implementation independence. The size of a page, 2^m, is chosen to be about as large but often somewhat larger than the page size that would be chosen on a linear paged address space system. If paging within objects is provided by making $n > m$, then the largest object on the system cannot exceed 2^n words. This limits the size of the largest object unless 2^n is greater than the total amount of physical memory (disk) on the system. This size, however, is approximately the same size as the entire address space on a linear paged address space system. Thus we see that mapping an ID space is more difficult than mapping a linear paged address space both because objects in the ID space are smaller than pages and because each entry in the auxiliary memory is larger for an ID space than for a linear paged address space.

An additional advantage of a paged linear address space is that it naturally allows storage to be swapped efficiently between levels of the memory hierarchy. The page size is chosen so that swapping will be efficient. If objects are very small, then many objects will be swapped at once while if objects are very large, they will be broken into pages. The address space does not place any limitation on the size of objects because the address space is larger than the total amount of physical memory on the system. Since most objects are quite small, it is essential that several small objects be swapped at one time. Although it would be possible to swap several objects at once on an ID space system, it would be necessary to keep track of the additional information of which objects were to move together, while in a linear address space this information is reflected in the relative placement of objects in the address space. Furthermore, the linear paged address space allows the operating system to manipulate pieces of storage of uniform size, thereby simply solving storage fragmentation problems for the operating system.

Figure 5 shows how several objects could be placed in ORSLA's linear paged address space. Page boundaries can appear at any place in an object. The programmer is not very concerned where page boundaries occur because ORSLA uses demand paging to bring those

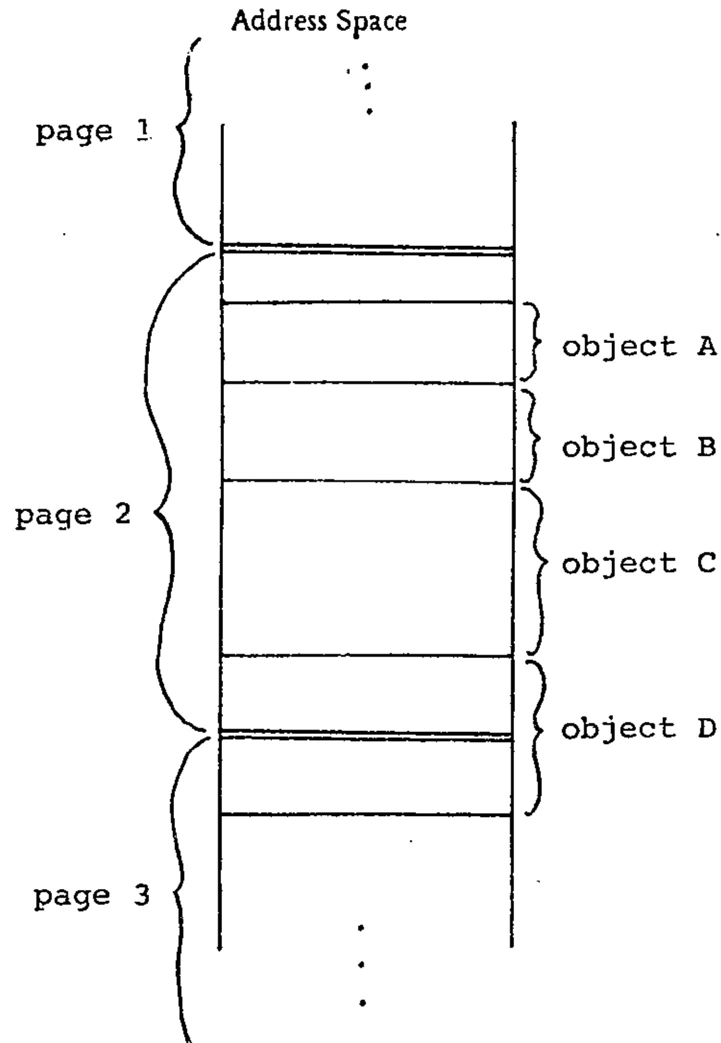


Fig. 5. Objects and Pages in a Linear Paged Address Space

---

words of the address space that are currently being used into high speed memory. Thus the page size on ORSLA is largely invisible to the programs on ORSLA. The page size on ORSLA is selected to make page swapping efficient. Some of the factors that must be considered in choosing the page size are: 1) access time of disk, 2) transfer rate of disk, 3) access time of high speed memory, and 4) size of high speed memory. Usually these factors result in page sizes of over 100 words. If not, however, the page size on ORSLA must still be larger than 100 words so the storage needed for the system catalog of pages will not be too large. The address space itself does not guarantee that swapping will be efficient. It merely allows swapping to be efficient if objects are placed in the address space properly. One obvious requirement for efficient swapping is that all or most of the address space in a page

be allocated to objects. Another requirement is that the objects that reside on the same page be used together much of the time. It is the responsibility of the programs and users on ORSLA to ensure that the objects on the same page will be used together. Since we don't want the programs or users on ORSLA to be dependent on the page size, however, we merely encourage them to place objects that will be used together adjacent to each other, as are objects A, B, C, and D in Figure 5. This technique will increase efficiency of swapping regardless of what the page size is or where page boundaries occur in the objects. The concept of *locality of reference* from virtual memory systems is important on ORSLA as well. A process exhibits locality of reference if its memory references during any one short time interval are concentrated in a few places in the address space rather than being uniformly spread over the address space. The placement of objects in the address space on ORSLA can greatly affect the locality of reference of the processes in the system. Swapping will be efficient if the processes on the system exhibit locality of reference.

3.2 Words

The address space for ORSLA is a linear address space consisting of words of virtual storage. Each w rd contains a *tag bit* that indicates whether the word contains an object reference or not. Fabry calls this the *tagged* approach to capability systems [Fabry74]. The tag bit allows the hardware to be aware of every object reference in memory.

The word size on ORSLA is chosen to be one bit larger than the size of an object reference. A word may be used for either an object reference or atomic data. Atomic data is any data that does not contain object references, such as character strings, bit strings, or machine instructions. Thus machine instructions in a program are considered to be atomic data rather than object references. Operand "addresses" that appear within machine instructions are actually offsets within objects that are pointed to by object references that will be in CPU registers when the instructions are executed. Although some of the cost of using an object reference is due to the time taken to perform memory accesses, this time is remarkably uniform for virtual memory systems. An important cost of using object references, especially for list processing, is the cost of copying and storing object references. Both of these costs are proportional to the size of the object reference. ORSLA can only

compete in the support of list processing with systems such as Multics that use bare addresses as object references if the object reference on ORSLA is less than twice the size of the address on ORSLA. As we will see, it will not be efficient to have fewer than 18 bits in the object reference for all the other fields combined, so if the size of an address is from 40-50 bits, then an object reference will have between 58 and 100 bits.

Object references must be aligned on word boundaries to prevent excessive overhead for tag bits[1]. Since most objects will contain at least one object reference, it is acceptable to require objects to begin on word boundaries by having the address within an object reference be a word address. The representation of an object, however, can use storage very efficiently by using the bits in the representation of the object for atomic data. The hardware will probably contain a field extraction unit that allows arbitrary size bytes to be extracted from the representation of an object. It will therefore be necessary to do bit addressing within an object. The address in an object reference is a word address, but the hardware can deal with bit offsets within an object. A bit offset from the beginning of an object does not count tag bits in the words of the object since the programmer cannot set the tag bits explicitly with byte manipulation instructions. In recent years it has become popular to make the word size a power of two. The only cogent reason for making the word size on ORSLA a power of two plus a tag bit, however, is to simplify the conversion from a bit offset within an object to the word offset within the object and the bit offset within this word. Since the hardware retrieves words, all accesses must be performed on words. The conversion from bit to word offset is achieved by dividing the bit offset by the number of bits per word minus one. The remainder from this division is the bit offset within the word. If the number of bits per word is a power of two plus one, however, e.g. 65, then the division by 64 is trivial because the quotient and remainder are obvious from the base two representation of the bit offset. If, on the other hand, the word size is guaranteed to be a small integer, i.e. less than 100, it is possible to provide several special purpose chips in the byte extractor that automatically divide by the word size in one step. Thus, if all the other

1. Redell discusses techniques for placing tags on object references that are not aligned on word boundaries.

considerations allow a range of word sizes that includes a power of two plus one, then the byte extractor hardware can be simplified and speeded up somewhat by choosing the power of two plus one as the word size. If other design criteria do not make a word size of a power of two plus one attractive, however, then a different size may be chosen.

3.3 Enforcing Restrictions on the Use of Object References

In Chapter 2 we saw that the essential feature of a capability system is that it enforces the restrictions on the use of object references that are inherent in the concept of objects. There are two features of objects that are very powerful for controlling access to an object. The first property, that the distribution of references to an object can be controlled by the software that created the object, is provided by the low level restrictions on the use of object references. The second property, that the software that defines an object can control the set of operations defined on the object and how these operations are implemented, is provided by the high level restrictions on the use of object references. In order to implement a high level abstraction, however, it is necessary for a low level abstraction for the object to exist as well. On ORSLA, the existence of high level and low level abstractions for the same object is achieved by having a *high-low* bit in the object reference. If the bit is *low*, then the representation dependent operations of reading and writing bits and object references from the representation of the object are defined. These operations are the load and store instructions on ORSLA. If the *high-low* bit is *high*, then none of these representation dependent operations are defined on the object. The only immediate operation that can be performed with a high level object reference is to set the *high-low* bit to *low*, but this can only be done by software that defines the data type of the object. Thus we see that enforcing the high level restrictions on the use of object references is largely a matter of controlling the setting of the *high-low* bit in high level object references. Enforcing the low level restrictions on the use of object references is much more complicated, however, so it will be considered first.

An object may be manipulated only through the use of a reference to the object. This restriction requires that storage may only be accessed through the use of an object reference. Thus the load and store operations must take an object reference as an operand. An offset

must also be given to the load and store operations to identify the word within the object that should be accessed. The offset may specify a bit string within the object that is to be accessed.

An object reference is created only when its corresponding object is created. The way to implement this restriction is first to prevent the creation of object references and then to provide some special primitives that create objects and the references for these objects. The creation of object references can be controlled by controlling the setting of tag bits. When a program stores an object reference into a location, the tag bit in that location should be set. When a byte of atomic data is stored, however, the tag bits of the words stored into must be set to zero. The tag bit is not under the direct control of the programmer: thus the tag bit is invisible to the byte manipulation instructions. The hardware ensures that the tag bits correctly specify where the object references are. Primitives that create objects and their associated object references will be discussed in Chapter 7.

Most modifications to an object reference are prohibited, especially modifications to an address within an object reference. Although the object reference is context independent, only a small part of the object reference always has the same interpretation: the type code. The rest of the object reference is part of the representation of the object, so the interpretation of these bits is dependent upon the type code. Thus a small integer uses these bits for the representation of an integer. Most types, however, need an address in the object reference. Both the type code and the address within an object reference are very sensitive fields. An address within an object reference cannot be modified at all, while the type code could possibly be changed to another type whose representation is similar to the representation of the original type of the object. Since the object reference must be protected against modifications, it is possible to have other fields in the object reference that must be protected as well. The rest of the bits in the object reference, however, are part of the representation of the object and may be manipulated and modified as is the rest of the representation.

Improper modifications to object references can be prevented by first preventing all modifications to object references and then providing specific primitives that perform only

the acceptable modifications. The general registers of the CPU on ORSLA contain object references, so every machine instruction must use these object references properly. Thus the add instruction operates only on objects that are numeric and the exclusive-or instruction operates only on objects whose abstraction is *bit string*. In addition, it is necessary to prevent manipulations of bytes within the representation of an object from modifying any object references in the object. Whenever a byte of atomic information is stored into a word, the tag bit of the word must be set to zero so that any object reference that was in the word and has now been modified can no longer be used as an object reference. Primitives to perform legitimate modifications to an object reference will be discussed in Chapter 7.

When an object reference is used to access storage, only the storage containing the representation of the object referred to may be accessed. This restriction is implemented in two parts. Some objects are able to hold their entire representation within the object reference itself. The references to such objects do not contain an address and so cannot be used to access storage at all. To allow the hardware to quickly identify such object references, there is a *data_type_info* field in the object reference. The first two bits of the *data_type_info* field are zero if there is no address in the object reference. Usually, however, an object reference does contain an address that points to a block of words in the address space. These words may be accessed by the load and store instructions which take two arguments: an object reference and an offset within the object. The offset is an integer which, when added to the address within the object reference, gives the address of the word to be accessed. On ORSLA, the address within the object reference points to the first word of the representation of the object. If the size of the representation is n words, then this procedure for finding the address to be accessed is only legitimate if the offset, i, satisfies: $0 \leq i < n$. Otherwise the address calculation described above will result in the address of a word outside of the representation of the object being accessed. Each memory access must therefore check the offset to ensure that it is within the proper range. The basic technique for reducing the cost of this check is to minimize the number of extra memory references needed to obtain the operands for this check. The offset being accessed, i, is already in the CPU. The only problem is finding what n is. If n is in the object reference, then it is also in the CPU. A separate piece of hardware can be added to the CPU to perform this check without slowing down the computation of the CPU. The memory access may proceed before

the check is complete so the check will not slow down computation at all. If the check is violated, the memory access can be aborted and, if necessary, any damage repaired.

Unfortunately, the need to have a *size* field in the object reference has caused object references on both the B6700 and the Rice-2 [Feustel72] to be more than twice the size of addresses on these systems. If there is no limit to the size of a single object, then an object may use all of the storage on the system which would require a *size* field of 34-44 bits depending upon whether the upper bound on online storage is 10^{12} bits or 10^{15} bits. Such a large *size* field would cause the object reference to be too much larger than an address for ORSLA to be able to compete with systems that use bare addresses. The purpose of the *size* field, however, is merely to reduce the time needed for checking the validity of loads and stores. It is not necessary for the *size* field to approve every valid load and store operation if more accurate size information is quickly available elsewhere. It is possible to encode the *size* field using between 5 and 9 bits so it would approve over 95% of the valid load and store instructions, thus reducing the overhead for checking the validity of loads and stores to less than 5% of the time taken for the load and store operations themselves. The exact method of achieving a small *size* field is rather complicated, however. The details will be covered in Chapter 7.

Although this covers the obvious way in which an object reference can be used to access storage in another object, there is another, more obscure way to violate this restriction. When an object, *x*, is destroyed, it is desirable for the address space to be reused for another object, *y*. If references to *x* still exist, then a legitimate use of a reference to *x* will result in accessing storage in *y*. This problem is so severe that operating systems have not allowed it to occur. It is a common problem in programming languages, however, where it is called the dangling reference problem. An object reference is dangling when the object it refers to has been destroyed. Dangling references only become a problem, however, when the address space for an old object is reused for new objects while a dangling reference to the old object is still accessible.

The dangling reference problem cannot be allowed to exist on ORSLA. Perhaps the best solution to the dangling reference problem is to wait until all of the references to an

object are destroyed or become inaccessible before destroying the object. There are basically two techniques for determining that an object is inaccessible: reference counts and garbage collection. ORSLA should support both techniques. Unfortunately both of these techniques are very difficult to achieve without much runtime overhead given all of the assumptions of ORSLA. The very large address space makes garbage collection difficult. This thesis deals with this problem. Chapter 4 will begin to discuss the problem of garbage collection on ORSLA. The cost of freeing storage for an object by garbage collection is proportional to the size of the object while the cost for reference counts is proportional to the amount an object is used[1]. Since most objects are small and/or are used a lot, most objects should be garbage collected. Reference counts should only be maintained on very temporary objects. The free copying of references, as in LISP, causes problems for reference counts. Each time an object reference is copied, it may be necessary to update a reference count as well. The problem of reducing the overhead for automatically maintaining reference counts on ORSLA will not be considered in this thesis, but it is a problem that must be faced before ORSLA can be built. Other workers [Deutsch76] have also recognized this as an important area of research.

The dangling reference problem can also be solved by simultaneously destroying an object and all references to the object. This can easily be built into the garbage collector that is needed on ORSLA. A slight variation on this theme is to free the storage for an object or group of objects without reusing the address space for these objects until all references to these objects have been destroyed. This also requires cooperation from the garbage collector. Thus ORSLA prevents the dangling references caused by the explicit destruction of objects from becoming a problem.

The last low level restriction on the use of object references is that *an object reference must always refer to the same object.* The most important mechanism that implements this

---

1. These estimates for overhead may surprise some people. The cost for garbage collection on ORSLA is derived in Chapter 6. If an object exists for a long time, then the major cost for reference counts is incrementing and decrementing the reference counts: operations that are performed automatically when the object is used.

restriction is the single, large address space. Dangling references violate this restriction when they become problems, so all the mechanisms needed to solve the dangling reference problem also implement this fundamental restriction on object references.

3.4 Enforcing the High Level Restrictions

The low level restrictions on the use of object references provide some protection, but their main goal is to ensure that an object reference actually points to the object it is supposed to point to. In addition, the low level restrictions ensure that the user who creates an object may control the initial distribution of references to the object. By itself, however, this does not provide much protection since to a large degree the distribution of references to an object is determined by the requirements for the flow of information in a computation. Thus if information about object A is needed when doing an operation on object B, there will probably have to be a reference to A in the representation of B. Usually the requirements of protection do not conflict with the requirements of legitimate computation, however, thus even if object B should not be able to obtain all of the information about A, it will probably be permissible for B to obtain the information it needs to have about A. The high level restrictions on the use of object references allow the operations that can be performed on an object to be limited, so the reference to A that is stored in the representation of B could be limited to those operations that allow only the information that B needs about A to be transferred to B. When combined with the ability of the creator of an object to control the distribution of references to the object, the ability to limit the operations that can be performed with individual object references forms a powerful protection mechanism.

Each object is a model of an ideal object known intuitively to the programmer; all of the operations on the object must be consistent with the ideal object being modeled. This is achieved by having the type code field in the object reference identify a data type definition that specifies what operations are defined on the object and how to perform these operations. As mentioned above, operations are forced to follow the data type definition by having a *high-low* bit in the object reference. If the bit is *high*, the representation of the object cannot be accessed, but the *high-low* bit can be set to *low* by the data type definition

which can then access the representation of the object to perform the required operation. The operation of changing the *high-low* bit from high to low is a very sensitive operation that can only be performed by the data type definition of the object. Exactly how this is enforced is discussed in Chapter 7.

Using the *high-low* bit to allow operations on an object to be limited has several effects on programming styles that I consider to be beneficial. First, the operations that can be performed on an object with a low level object reference cannot be limited, so a high level object reference must be used whenever the operations on the object need to be limited. This encourages the programmer to limit the high level operations on the object rather than the low level operations. Sophisticated protection is expressed as limitations on the high level operations, however, so this effect on programming is beneficial.

An important property of high level operations in programming languages and data base management systems is that high level operations are representation independent, i.e. they operate on any object whose high level abstraction is appropriate regardless of how that abstraction is represented. Thus a program that uses protection via the *high-low* bit also achieves representation independence. Conversely, modules that want to achieve representation independence also achieve an extra measure of protection. Often the use of high level operations as well as the need for sophisticated protection and representation independence occur at module boundaries. If module A passes the object x to module B, module A will want module B to perform high level operations on x and may want to limit the high level operations that B may perform on x. Module A may also want to reserve the right to change the representation of x or to create another representation of the same kind of abstract object as x without having to recompile module B. By combining protection and representation independence with high level operations, module B is forced to use x in a representation independent manner. Thus if modifications are made in A that do not affect the abstraction that x presents to B, then B need not be recompiled. I consider this to be a beneficial effect on programming.

The difficulty with limiting operations only by using the *high-low* bit is that it is difficult to protect an object from representation dependent code. The only representation

dependent code that can operate on an object, however, should be in the data type definition for the object, thus this limitation is merely that an object will not be protected from its data type definition. The entire data type defintion must be correct, however, in order for the object to achieve the behavior it is supposed to have. Other modules that only use the high level abstraction of the object are allowed to depend upon the object achieving the behavior it is supposed to have. These modules may be proven correct on the assumption that this object behaves as it is supposed to behave. These proofs will be meaningless if any part of the data type definition of the object is incorrect, thus little is gained by protecting an object from its data type definition. By placing the programmer on notice that the entire data type definition must be correct, however, we may be able to encourage the programmer to prove the correctness of the data type definition, thereby increasing overall system reliability.

Programming languages are satisfied with two abstractions for an object: high level and low level. In order to provide protection, however, it is necessary to be able to support many abstractions of the same object. Each abstraction has its own set of operations that may be performed on the object. One abstraction is needed for each kind of access that is to be provided to an object. Often the sets of operations associated with two abstractions for an object will overlap a good deal, but there is no need for them to overlap at all. In order to support multiple abstractions of the same object it is useful to have a field in the object reference that specifies which abstraction is being used by this reference. This field is called the access control field because it will be used to control access to objects, although it can be used for any purpose that requires multiple abstractions of the same object. The upper limit on the size of the access control field is determined by the need to keep the object reference small. Up to now, capability systems have used a very inefficient way of coding the access control field by using each bit to represent the legality of a different operation, so the access control field could specify an arbitrary subset of the possible set of operations on the object. This coding is not feasible for high level abstractions for which there may be a large number of operations defined on the object. There has not been adequate experience with more efficient codings in which the entire access control field is an integer that identifies a useful abstraction. We can only guess how many bits are needed in the access control field. My guess is that between 4 and 10 bits are needed.

3.5 Format of the Object Reference

The object reference on ORSLA contains several fields. Perhaps the most important field is the type code field. If the object reference supported by the hardware on ORSLA is to be used by all the subsystems on ORSLA, it must be possible to add new types of objects to ORSLA. This is only possible, however, if there are a sufficient number of undefined type codes in the object reference. Since the type code is supposed to identify the data type definition, however, it is tempting to use the address of the data type definition as the type code. Unfortunately, this causes the object reference to be more than twice the size of an address, which is unacceptable. Furthermore, there is no need for 2^{40} data type definitions since they would not all fit in the online storage on ORSLA. The size of the type code field can be minimized by placing an object reference to the data type definition into the representation of the object. This solution is not acceptable either, however, because it increases the size of objects by one word and requires a memory access to find the type of an object. A third alternative that compromises between these two solutions somewhat while keeping a small type code is to provide a type code field that is large enough to identify the data types that are provided initially by the system and whose definitions may be implemented in hardware. One of the initial data types is the *escape* data type that contains an object reference for its data type definition in the first word of the representation of the object. The Rice-2 was able to achieve a 5 bit type code field using this approach while still allowing an unlimited number of data types. If it is advantageous to use the type code field to identify the data type for frequently used system defined data types, then it is also advantageous to identify data types that have been defined by subsystems and are used heavily in those subsystems directly in the type code field. If the user can define a data type that is identified by the type code in the object reference, then the problem arises of why the user should use *escape* data types. The answer, of course, is that unallocated type codes are a scarce resource of the system that should be carefully allocated, since a type code cannot be reused once it has been defined. The standard techniques for discouraging the use of scarce resources, such as quotas and charging for the use of the resources can be used for type codes. If it costs a user $1000 to allocate a type code, users will not allocate type codes unless they are sure it will save more than $1000 of storage and/or CPU time. Neither quotas nor charging for the use of the resource can handle proper allocation of an extremely scarce

resource, however, so it is necessary to make the type code field large enough so these mechanims will not use up all the type codes. A 9 bit type code field is about the minimum size that will allow most subsystems to use a small number of type codes without exhausting the supply. As the type code field becomes larger, however, there begin to be problems converting a type code into the address of the associated data type definition. This is a serious problem once the type code field is as large as 16 bits. Thus the type code field should be in the range of 9-16 bits.

The type code field and the *size* field are both needed to reduce the runtime overhead and the storage overhead in small objects. There is also a need for another small field in the object reference called the *data_type_info* field. This field contains information about the representation of the object that allows certain operations to be performed on the object reference without accessing the data type definition at all. For example, this field specifies whether the object reference contains an address or not. It also specifies whether reference counts are being maintained on this object or not. It will not be known exactly what this field will be used for until the design of ORSLA is complete and construction is ready to begin. Thus, this thesis does not specify all of the information that is contained in this field. The information that I suggest should be contained in the *data_type_info* field is described in detail in Chapter 7. I estimate that the *data_type_info* field will use between 3 and 5 bits.

An interesting possibility for reducing the size of the object reference is proposed in this thesis. The *size* field is only needed in low level object references, while the access control field is only needed in high level object references. Both of these fields are approximately the same size (5-9 bits), so I propose that the same set of bits in the object reference be used for these two fields. The operation of converting a high level object reference to a low level object reference will extract the access control field and will load the *size* field with valid information. Similarly the operation that converts a low level object reference to a high level object reference will load the access control field. The details of these operations will be covered in Chapter 7.

We have now seen all of the fields that exist in the object reference on ORSLA: the type code field, the *data_type_info* field, the *high-low* bit, the access control field, the *size*

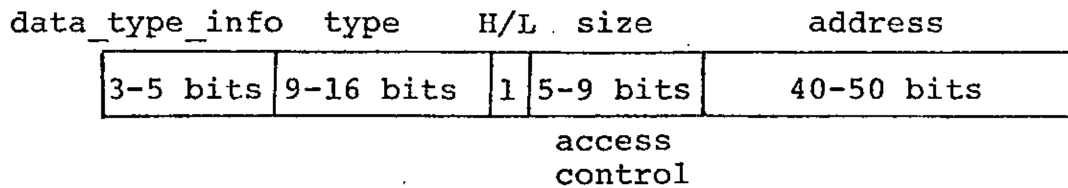


Fig. 6. Format of the Object Reference

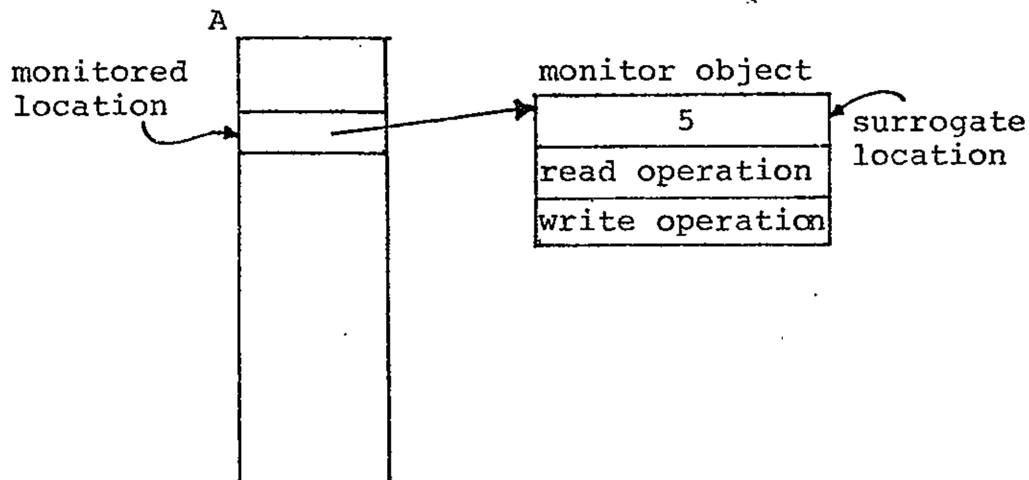
field and the address. The size of the address is 40-50 bits, but the size of the object reference must not be more than twice the size of the address in order to allow ORSLA to compete with computer systems that do not support context independent object references in the hardware but manipulate addresses instead. We have seen that the type code field requires 9-16 bits, the *size* field requires 5-9 bits, the *high-low* bit requires one bit, the *data_type_info* field requires 3-5 bits, and the access control field can use the bits in the *size* field. This gives a total range of 18 to 31 bits in the object reference in addition to the address. We cannot really afford more than 31 bits in the object reference for fields in addition to the address, however, because 31 bits is so close to the size of the address. Thus the size of the object reference on ORSLA will be in the range of 58-81 bits.

3.6 Monitoring

There is an interesting feature on ORSLA called "storage monitoring" that allows all accesses to a particular storage location to be monitored. Monitoring the value of variables is an accepted control structure in high level languages [Fisher70, pp98-99; Hewitt72]. Many languages only support monitoring of simple variables because otherwise monitoring causes a great deal of overhead if it is not supported by hardware. If monitoring were provided in the hardware, however, there would be almost no overhead until a monitor were actually "tripped" or invoked. It would be possible to monitor not only simple variables, but also elements in arrays and other data structures. This kind of monitoring can be used to provide a trace of the values of a variable and/or to ensure that the value of a variable always follows restrictions that the programmer has specified. These are both powerful debugging tools. The general storage monitoring feature is also similar to the

"stop-on-address-compare" switch that makes hands-on debugging using the "single-step" switch and other console switches so powerful. Storage monitoring can also be used to accumulate metering information without including special metering instruction sequences throughout the system software. Thus system evaluation can be done much more easily with storage monitoring. It is also possible to detect the use of uninitialized variables with storage monitoring without inserting explicit checks into the software. Finally, storage monitoring can be used to prevent the modification of certain sensitive fields in the representation of an object.

Storage monitoring is implemented by replacing the contents of a monitored location by a special object reference in which a bit in the *data_type_info* field specifies that the location is monitored. The object reference stored in the monitored location points to the monitor object. Whenever a monitored location is accessed, the definition of the monitor object is inspected to determine how the read or write operation is to be performed. The



A location in object A which is supposed to contain an object reference to the integer 5 is being monitored. Whenever this location is accessed, the definition for the read or write operation that is within the monitor object is invoked. The *storage_monitor* bit in the *data_type_info* field in the object reference in the monitored location is on. This bit activates the storage monitor machinery during reads and writes.

Fig. 7. General Purpose Monitor Object

most common storage monitor object contains three locations as shown in Figure 7. The first location is a surrogate location that contains what the monitored location would contain if it were not monitored. The second location contains a reference to the definition of the read operation. If this is the *null* object, then a read from the surrogate location is performed, otherwise the definition of the read operation must be a procedure object that is called with two arguments: the current monitor object and a cell object that specifies the monitored location. The value returned by this procedure is considered to be the information read from the monitored location. The procedure may perform any computation that is desired. The cell object allows monitors to be added to and/or removed from the location and allows read and write operations to be performed to the monitored location. The third location in the standard storage monitor object contains a reference to the definition of the write operation. This is either null or a procedure that takes three arguments: the monitor object, the cell object, and the information being stored into the location. The write operation, of course, does not return a value.

Another common storage monitor object is the *uninitialized_storage* monitor object. Whenever an object is created on ORSLA, all the storage in the representation of the object is automatically initialized with *uninitialized_storage* storage monitors. The reference to the *uninitialized_storage* object is an atomic object reference that contains no information. The store operation to an uninitialized location is defined to erase the storage monitor and perform the store operation normally. The read operation to an uninitialized location, however, is defined to causes the *uninitialized_storage* interrupt. Software may set up a handler for this interrupt, but most of the time it will generate an error.

The object reference for the monitor object that is passed to the procedure that defines the read and write operations for the most common storage monitor object is the same as the object reference in the monitored location except that it has been *deactivated* as a storage monitor by having the *storage_monitor* bit in the *data_type_info* field turned off. Thus this reference to the monitor object can be passed from program to program, allowing the representation of the monitor object to be accessed, while an active reference to a monitor object causes an interrupt that invokes the monitor whenever the location containing the active reference to the monitor is accessed.

Whenever a read or write operation is performed to any location, the contents of the location must be checked for the presence of a monitor. If the *storage_monitor* bit is chosen as part of the system design, then a gate can be placed in the hardware that will generate an interrupt when a location containing an active storage monitor is accessed. Most read and write operations will be performed on locations that are not monitored, so the checks for monitors must be designed to optimize the case of no monitors. With the combinatorial circuit mentioned above, the read operation can check for monitors without taking any extra time. When a location is written into, however, the previous contents must be checked for a storage monitor. This requires that the previous contents be accessed before the location is modified. It is not necessary to use a full read-modify-write cycle on the memory, however, since a monitor will be found so seldom. It is acceptable to use a read/write cycle in which the information written is not dependent upon the information read. If a location is monitored, then this will only be discovered after the write, but then the original contents can be rewritten, thus forming a read/write-rewrite cycle that is indistinguishable from a read-modify-write cycle in which the write has been aborted. These complex memory operations require a little more time than a simple write cycle on semiconductor memory. Unfortunately, it is difficult to estimate just what this overhead is. The necessary read/write cycle can be just as fast as a simple write cycle, as on Intel's 1103 family of 1K dynamic MOS RAMs, or it can be as long as 190% of a simple write cycle as on Signetics N82S11 bipolar 1K RAM. The architectures with the highest speed memory, however, make use of a cache. As we will see in Chapter 5, cache memory must perform a read before the write anyway. Monitoring is able to make use of the data thus retrieved during store operations rather than wasting it. Thus it is possible that a small overhead will be incurred in order to implement monitoring on slower, less expensive machines, but the fastest machines will not incur any overhead for this feature.

There is another use for the checks needed by storage monitoring, however: they can be used to implement reference monitoring as well. Storage monitoring monitors the use of a location while reference monitoring monitors the movement of an object reference. The most important use for reference monitoring is to maintain reference counts. Most objects will have their storage reclaimed by garbage collection, but some objects will also make use of reference counts. Since the number of objects that will make use of reference counts is

significant, the maintenance of reference counts will probably be done in microcode. A bit in the *data_type_info* field in the object reference will be used for identifying those references for which reference counts are being maintained. Another input will be added to the gate needed for storage monitoring so it will recognize reference monitors as well.

In order to maintain reference counts it is necessary to perform a few more checks than are performed for storage monitoring; since these extra checks all involve information that is already in the CPU, however, they can be performed without slowing down the CPU. Whenever the number of references to an object changes, the reference count must be updated. A load operation makes a copy of the reference stored in the location accessed, so the count on the object referenced must be incremented, but the reference is stored in a CPU register which may have held a reference to another object. The number of references to this second object has decreased, so its reference count must be decremented. Thus not only must the contents of the location loaded from be checked for monitors, but so must the CPU register being loaded into. In the case of a store operation, the previous contents of the location must be checked. If it is an object reference, the reference count on this object must be decremented. In addition, it is necessary to check the object reference in the register being stored and the reference count on this object must be incremented.

Since checking for monitors does not cause overhead when no monitors are present, the overhead for reference counting is restricted to those objects for which reference counts are being maintained. Each time a reference to a reference counted object is copied or destroyed, the reference count, which is located in the first word of the representation of the object, must be accessed so it can be incremented or decremented. This extra memory reference whenever an object reference is moved is the minimum overhead that can be obtained if reference counting is performed by hardware. Even so, this overhead is very high when it is considered that on ORSLA object references are moved as frequently as pointers are moved within PL/I or LISP. Thus reference counting will only be used for very temporary objects whose references will not be copied much before their storage is reclaimed. It may be possible to reduce the overhead for reference counting by reducing the number of operations for which it is necessary to increment or decrement the reference count. For example, it may be possible to keep track of those references internal to the CPU separately

from those outside, thus bringing a reference into the CPU would cause the reference count to be incremented and when the reference is no longer in the CPU the reference count would be decremented, but the number of copies of the reference within the CPU would not be maintained in the object's reference count. In this case, the extra check on the load instruction may become unnecessary, although the extra check on the store instruction is still necessary. Such a technique would substantially reduce the overhead for reference counting for register-to-register operations. Exactly how reference counts should be maintained, however, is beyond the scope of this thesis. It is a subject that obviously needs further work.

Once all of these checks are provided for, some data types should be designed that can be used for a general reference monitoring feature. A comprehensive design of a monitoring mechanism and of language constructs to handle it is beyond the scope of this thesis. Providing such a comprehensive system is important for any system with tagged architecture such as ORSLA. Storage monitoring and the mechanism needed for the automatic maintenance of reference counts are very similar, so these two mechanisms should be combined into a more powerful and comprehensive monitoring feature. This is a fertile area for future research.

3.7 Small Objects

One of the important contributions of this thesis is to show how to swap many objects between high speed memory and disk in one disk operation. The first step in achieving this on ORSLA is the paged address space which allows many objects to be placed on one page. The next step is to provide a mechanism for encouraging locality of reference. Objects that will not be used together should not be placed on the same page, while objects that will always be used together should be placed on the same page.

ORSLA provides a mechanism for partitioning all of the objects on the system into groups called areas. Objects in the same area are supposed to be used together while objects in separate areas are less likely to be used together. Since placement within the address space determines placement on pages, areas must be used to place objects into the address space so that related objects can be on the same page. The area is therefore used to allocate storage

for the representations of objects[1].

Address space is allocated to areas in units of a page since the purpose of an area is to group objects together that are likely to be used together. Each area may use its own storage management techniques to allocate address space to individual objects. The system may allocate pages of physical storage to an area separately from the pages of address space. A serious problem that every system must face is the possibility that a single computation will exhaust all available physical storage. If the space hungry computation is running correctly, this is a serious problem indeed for which there are no easy answers. If the computation has exercised a bug that is causing the computation to allocate large amounts of storage uselessly, the computation must be stopped before all available storage is exhausted since the entire system may crash at that point. This can be prevented by placing storage quotas on areas. An erroneous computation will probably violate a storage quota long before all free storage is exhausted. The storage quota violation will stop the erroneous computation and allow it to be debugged. Areas must also have address space quotas since pages of physical storage and pages of address space can be allocated to areas separately.

When an object is created on ORSLA, it is necessary to specify in which area to place it as well as what kind of object to create. If the scheme for areas provided by ORSLA makes sense to the programmer then it will be easy for him to select which area to use. By specifying in which area to place an object, the programmer is giving the system additional information about the pattern of use of the object.

Regardless of how carefully objects are originally placed, however, as data structures change it may become necessary to move objects from one area to another. Once an object has existed for awhile the pattern of references to the object from elsewhere in the system provides important information about the best placement of that object. All of this information is available to the system. In fact, this information is used during garbage collection. On ORSLA, garbage collection is also used to increase locality of reference,

---

1. The term *area* comes from the PL/I area..

especially within a single area. It can also be used to move objects from one area to another. The resulting automatic mover is capable of correcting some mistakes in the initial placement of objects as well as recognizing changing data structures and moving objects accordingly.

3.8 Garbage Collection

Once it was decided that ORSLA must have garbage collection, many applications arose that could take advantage of garbage collection. It is essential that garbage collection be practical on ORSLA. Garbage collection operates by finding all accessible objects, but it is clear that it is impractical to find all of the accessible objects in 10^{12} bits of storage.

An important contribution of this thesis is to show how to garbage collect a single area separately from the rest of the system. Only the accessible objects within one area are found during a single garbage collection. The size of an area can be controlled so there will not be too many objects in the area to garbage collect it all at once. The greatest efficiency gained by this approach lies in the ability to garbage collect only the portions of the system that need garbage collection.

The key step in being able to garbage collect a single area is in having ORSLA maintain lists of all inter-area references. The garbage collector, when finding all the accessible objects in an area, can assume that objects referenced from other areas are accessible. This assumption does not always hold, however, so a mechanism is presented in Chapter 6 that prevents this assumption from causing problems when it does not hold. Chapter 4 begins the detailed description of the garbage collector used on ORSLA.

The real problem in ORSLA, however, lies not in the garbage collector itself but in the mechanism that maintains the lists of inter-area references. It would be easy to create such a mechanism that would cause a large amount of runtime overhead. The mechanism presented here for maintaining the lists of inter-area links would have to be designed into the hardware; it would then cause only a small amount of runtime overhead. This mechanism is described in Chapter 5.

Chapter 4
Garbage Collection in Areas

This chapter discusses garbage collection and the mechanisms that need to exist in areas in ORSLA in order to allow a small number of areas to be garbage collected. The mechanisms of inter-area links and of the lists of inter-area links are automatically maintained by ORSLA to allow small garbage collections. Cables and local computation areas prevent inter-area links from being generated so frequently on ORSLA that it would significantly slow down the entire system. The concept of inter-area links and how they are used to achieve small garbage collections is relatively straightforward, however. Cables and local computation areas are important for achieving efficiency, but are reasonably obvious mechanisms once the approach of inter-area links has been taken. What makes this thesis so novel even though it is based on a simple idea is the efficiency of the described mechanisms. Though it may seem that maintaining the lists of inter-area links would be very expensive, it turns out that this is not true, although we will not see why until Chapter 5. If the maintenance of the lists of inter-area links is carefully built into the hardware, it can take advantage of the virtual memory machinery by placing into the page map a reference to the area that each page is part of. The operation of finding what area an object resides in then becomes very inexpensive, thereby reducing the overhead for the maintenance of inter-area links to an acceptable level. The role of this chapter is to show the value of the lists of inter-area links so that the reader will appreciate the mechanisms for maintaining the lists of inter-area links that are presented in Chapter 5.

4.1 Overview of Garbage Collection

The main purpose of garbage collection is to free the storage occupied by inaccessible objects. It is not possible, however, to discover directly which objects are inaccessible. Instead a garbage collector operates by finding all of the accessible objects. All objects that are not accessible are by definition inaccessible. The inaccessible objects are freed and storage is compacted.

The accessible objects are found by a process called *marking*. Marking begins with the immediately accessible objects: the activation record of the procedure that called the garbage

collector, the file system, and all global variables. Logically two sets are created: the set of marked objects, M, and the set of objects that have been marked from, F. M initially contains the immediately accessible objects while F is initially empty. A marked element, x, which has not been marked from, is selected from the set M - F and is marked from. This is done by looking at the representation of x and finding all the object references in x. All these objects are added to M without causing duplications. The element x is then added to F. This process is repeated until M = F, i.e. M - F = 0. At this point M is the set of all accessible objects; marking is completed. The storage for all the inaccessible objects is then freed. There may be a fair amount of fragmentation of storage, however, i.e. there may be many small blocks of free storage separated by accessible objects. Compaction rearranges all the accessible objects in storage so that all free storage is in one large block. On systems with virtual memory, compaction also attempts to increase the locality of reference of the accessible objects. Note that when an object is moved in the address space all of the references to that object must be modified. Compaction must handle all these details.

4.2 Copying Garbage Collection

The basic purpose of a garbage collector is to clean up all the "garbage" that degrades system efficiency. This can be done by reclaiming storage for inaccessible objects, eliminating storage fragmentation, increasing the locality of reference of the data, and changing the representation of some objects, e.g. changing the size of a hash table. Some garbage collection algorithms can only do some of these tasks. An algorithm that will perform all of these tasks is the copying garbage collector [Fenichel69]. There are two phases to a copying garbage collector: the copy phase and the delete phase. The copy phase performs marking and compaction while the delete phase performs freeing.

During the copy phase a copy is made of each accessible object. The object references placed in the new copy of an object are references to the new copies of the corresponding objects. When the copy phase is over, all of the accessible objects have been copied. Since all these objects were allocated from one large block of free storage, there is no fragmentation in the copy. Also, there are no object references from the new copy to the old copy.

The delete phase then deletes the old copy. All of the storage in the old copy, whether

it was part of an accessible object, an inaccessible object or free storage, is freed at once as one large block. No information about the representation of inaccessible objects is necessary if the freeing is done in this way. Most other implementations of garbage collection require such information.

The copy phase of the garbage collection must maintain a mark data base. All objects that have been marked have an entry in the mark data base that contains a reference to the new copy of that object. There are two ways of maintaining this data base: by internal marking or by external marking. External marking uses a separate mark data base that associates marked objects with their new copies. Internal marking uses a word in the representation of the marked object to hold a reference to the new copy of that object. During the garbage collection, most of the objects are dealt with by the garbage collector as objects that are being garbage collected. A few of the objects are also used to assist with the garbage collection, however, such as the garbage collection procedures. If the object is not used to assist with the garbage collection, however, it is not necessary to reserve an entire word in the representation of the object for internal marking. It is possible to reserve only a single bit solely for internal marking by having this bit indicate whether the object is marked or not. If the object is marked, then a specific word of the representation of the object contains a reference to the new copy of the object instead of the part of the representation of the object that the word contains when the object is not marked. Objects that are internally marked in this way have part of their representation obliterated when they are marked, and so cannot be used while they are marked to model the object they are supposed to model. If an object may be used to assist with the garbage collection then an entire word in the representation of the object must be reserved for internal marking. External marking, on the other hand, does not use any storage within individual objects. It allows all objects to be used to assist with the garbage collection as long as their state is not modified by the garbage collector. Whether internal or external marking should be used is immaterial to this thesis. Since external marking is easier to work with, however, it will be assumed.

The copying garbage collector begins with a list, L, of immediately accessible objects. The heart of the copy phase is in the recursive procedure *collect* shown in Figure 8. The

collect procedure is applied to each of the elements of L. This ends the copy phase. The language used in Figure 8 is modified Algol that allows a variable to be declared to be an *object*, i.e. the variable may be assigned any type of object. Variables of type *object* use a full context independent object reference. The assignment statement only involves the movement of an object reference, it does not cause information from within the object being assigned to be copied into any other object. The *externally_mark* procedure adds an entry to the external mark data base, while the *externally_marked?* and *new_copy* procedures retrieve information from that data base. The ith word in the representation of x is specified by $x[i]$.

The code in Figure 8 merely describes the basics of the *collect* procedure. It is not code that can be executed. Most of the procedures called by *collect* are dependent upon the representation of x. I am not suggesting that these specific operations should exist on x. Probably *collect* should be written separately for each data type. The *collect* operation would be the generic operation defined on all data types to perform garbage collection. The code in Figure 8 merely describes what this operation would do.

```
procedure collect(x);
    begin object x, new_x; integer i;
      if externally_marked?(x) then return new_copy(x);
      new_x := allocate_new_copy(x);
      externally_mark(x, new_x);
      for i := 1 to size(x) do
            begin object z, w;
            GC_load (z, x[i]);
            if storage_monitor?(z)
                then w := deactivate_monitor(z);
                else w := z;
            if ~atomic?(w) then w := collect(w);
            if storage_monitor?(z)
                then set_storage_monitor(new_x[i], w);
                else new_x[i] := w;
            end
      return new_x;
    end
```

Fig. 8. COLLECT Procedure

The *collect* procedure takes one argument and returns one value. The argument is the object to be garbage collected and the value is the new copy of the object. If an object has already been marked then there will be an entry in the external mark data base specifying where the new copy is. The new copy can be immediately returned as the value of *collect*. If the object has not been marked before, then a new copy must be created and an entry must be made in the external mark data base. Then we may begin putting information into the new copy. Information that does not contain addresses (atomic information) can be simply copied into the new copy of the object. The test for atomic information is true if the tag bit is zero or if the type code indicates an atomic object reference, i.e. the first two bits of the *data_type_info* field are zero. Non-atomic object references cannot be copied so easily. An ordinary non-atomic object reference must be converted to a reference to the new copy of its object. The *collect* procedure will find the references to the new objects, so it is called recursively and the result is stored in the new copy of the object. This recursive call to *collect* not only finds the references to the new objects, but also causes objects that have not yet been copied to be copied. During the course of the garbage collection all accessible object references are passed to the *collect* procedure as arguments.

The existence of storage monitors does cause some complications for the garbage collector. The garbage collector should not trigger storage monitors or cause them to disappear. Thus when the information in an object is being copied, a special load instruction (*GC_load*) must be used that will not trigger storage monitors but will allow them to be copied. Thus the *GC_load* instruction returns an active object reference to the storage monitor. The *collect* procedure can determine whether the location was monitored by inspecting the *storage_monitor* bit in the *data_type_info* field of this object reference. Before the monitor object itself can be copied or manipulated in any way, however, the reference to the monitor object must be deactivated. The *deactivate_monitor* operation turns off the *storage_monitor* bit in the *data_type_info* field of the reference to the monitor object so that the reference to the monitor object can be passed around without triggering the monitor. After the monitor object has been *collect*ed, the reference to the new copy of the monitor object can be placed as an active storage monitor in the new copy of the object that was monitored by using the *set_storage_monitor* operation. The *set_storage_monitor* operation is basically a store instruction that turns on the *storage_monitor* bit in the *data_type_info* field

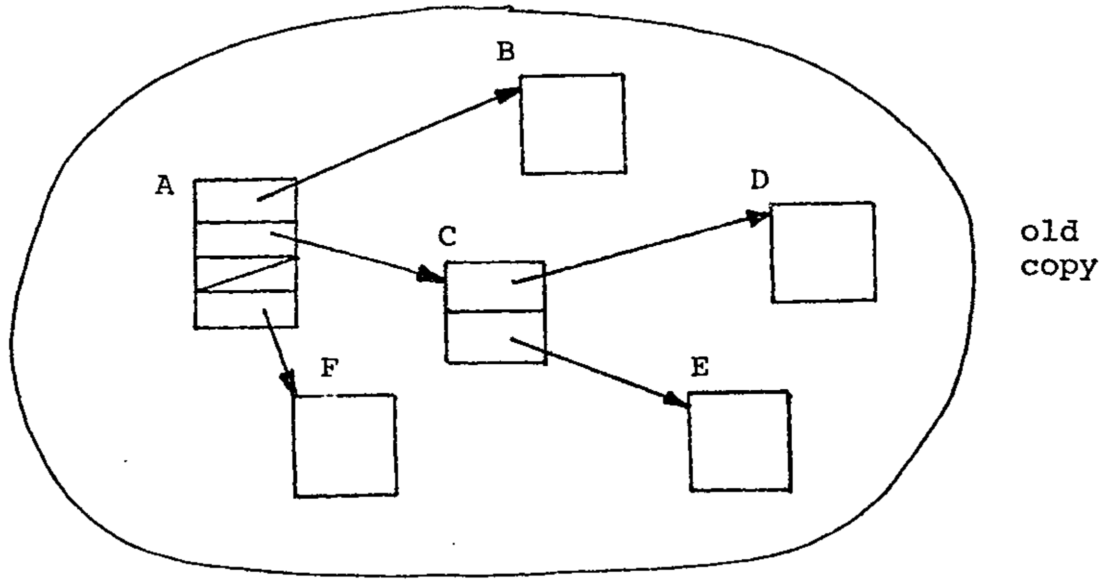
of the object reference being stored.

Note that each activation of *collect* that makes a recursive call is marking from one object. If the new copies are being allocated from one large block of free storage, then *collect* will place an object near other objects that reference it. For example, consider Figure 9. Assume that none of the objects A-F have been marked until *collect*(A) is executed. Regardless of where objects A-F are placed in the old copy, they are all placed adjacent to each other in the new copy. During the processing of this data structure three recursive calls on the *collect* procedure were active at one time. A simple copying garbage collector not only handles fragmentation of storage but automatically places together objects that reference each other and are therefore likely to be used together. This has been described by Fenichel & Yochelson [Fenichel69].

4.3 Large Garbage Collections

Garbage collection has been developed primarily as a programming language feature in which only the temporary storage for a single computation is garbage collected at once. This corresponds to the entire address space on an SAV system. Unfortunately, garbage collection becomes inefficient when used in much larger address spaces as on ORSLA. If paging activity is ignored, then the time required by a garbage collector is proportional to the amount of storage in the accessible objects. If the amount of storage in the address space being garbage collected is less than the size of high speed memory then the working set of the garbage collector will probably fit in high speed memory and the time taken for paging activity will be minimal. Finding all of the accessible objects in the address space is feasible if the size of the address space is smaller than high speed memory. If the address space covers all of disk, however, such a garbage collection is totally impractical. The garbage collection would probably thrash, but even if it did not there are so many accessible objects on the disk that looking at them all is a very large job. Garbage collection cannot be performed in exactly this way in ORSLA's single address space.

There are two sources of additional overhead that appear when garbage collecting large address spaces. First, the garbage collector thrashes, greatly increasing the amount of CPU time needed per accessible object. Second, much of a very large garbage collection is



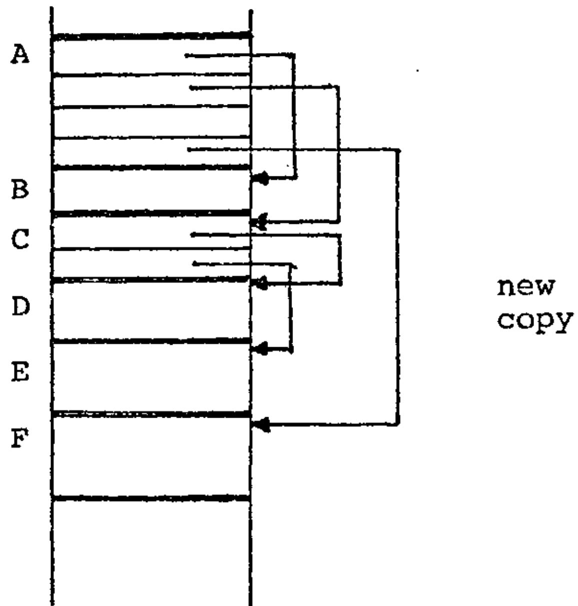


Fig. 9. Copying Garbage Collection

wasted on data structures that have not been modified since the last garbage collection. Just as a system exhibits locality of reference, so it exhibits locality of modification (see section 6.7). If the garbage collector could be concentrated on the parts of the system that are being modified, then most of the wasted garbage collection time could be avoided.

Both of these problems can be attacked by a garbage collector that operates on just a small piece of the address space. Objects on ORSLA, however, are already grouped together in units called *areas* in order to efficiently swap small objects in the memory hierarchy. Each area contains a small percentage of the storage on the entire system. It would be possible to garbage collect a single area if the system were able to list all of the object references from the other areas in the system to objects in the area being garbage collected. Then it would be known what objects within the area are accessible from the rest of the system. Using these as immediately accessible objects, garbage collection would find the rest of the accessible objects in the area. ORSLA provides such a mechanism; it automatically maintains lists of all of the inter-area references on the system so that areas may be garbage collected separately from each other.

Savings are achieved by this approach in three ways. First, the storage in the system can be garbage collected one piece at a time, allowing reasonably small garbage collections to be interleaved with normal computation. In addition, only the computations that are using the storage being garbage collected must be stopped during a garbage collection. Second, areas can be made small enough so that the garbage collector will not thrash, thereby significantly reducing the time needed to garbage collect a page of storage. Third, the garbage collector may be directed to those areas that need it the most. In particular, those areas in which modifications are being made at the highest rate will need to be garbage collected at a higher frequency than other areas. The areas that need a high frequency of garbage collection may have it without forcing other areas to be garbage collected at the same rate.

In Knuth's discussion of garbage collection, he states that garbage collection is performed when the system is about to run out of storage. Although this has been true on systems from the IBM 709 to the PDP-10 this is not true on large virtual memory systems

such as Multics [Fenichel69]. LAV systems only run out of storage when the entire system exhausts disk storage. A computer system should have large quantities of disk storage available for temporary storage at all times. If it is exhausted, no processes will be able to use additional temporary storage: an unacceptable situation. On ORSLA, an area is garbage collected when it contains enough garbage so that a garbage collection will save money by reducing either the storage used or the paging activity of the area. The longer the area will exist without modifications, the more money will be saved for each word of storage retrieved and for each increment in locality of reference achieved by the garbage collection.

Since each area contains a small part of the storage on the system, and since a computer system keeps a significant amount of disk available for temporary storage, the garbage collector does not have to be painstakingly designed to use a minimum amount of temporary storage. Thus we may use a copying garbage collector with a recursive collect procedure and an external mark data base. Rather than minimizing the total amount of temporary storage used, we should minimize the working set of the garbage collector. External marking may result in a smaller working set than internal marking.

4.4 Other Approaches to Large Garbage Collections

HYDRA shares many of the goals of ORSLA but tries to achieve them with a CUID system. HYDRA performs garbage collection on the entire address space at once to free storage for objects. This garbage collection, however, involves relatively large objects in a system with a small total of online storage. Even so, the garbage collection is so expensive that it is only done once a day. HYDRA requires 3 minutes to garbage collect 3×10^8 bits (40 million bytes) of online storage. ORSLA, however, is being designed for 10^{12} bits of storage. Ignoring non-linear effects due to thrashing, this would require at least 10^4 minutes to garbage collect, or approximately one week. If HYDRA had 10^{12} bits of storage it would not be able to perform garbage collection. The vast amounts of storage that systems can now have rule out the use of algorithms that must operate on all storage.

Recently there has been interest in having the garbage collector run in parallel with normal computation [Steele75, Wadler76, Baker77]. One reason for interest in this algorithm has been the hope that it would make garbage collection of all of disk storage practical.

Although a parallel garbage collector may allow processes to be garbage collected with a minimal interruption of service, it will not make garbage collection of all of disk storage practical. Even ignoring the high speed memory used by the garbage collector and the overhead for coordinating the running computation and the garbage collector that is required by Steele's algorithm in both the running computation and the garbage collector, it would take the garbage collector one week to complete a single garbage collection of 10^{12} bits of storage. The system would need enough extra physical storage to contain all the garbage generated on the system in a week.

Performing garbage collection in parallel with normal computation will not eliminate the need to be able to garbage collect single areas; on the other hand, once the decision has been made to support garbage collection of single areas, it may be desirable to support garbage collection of a single area in parallel with use of the area. Unfortunately, there has not been enough analysis of the overhead involved in parallel garbage collection to enable me to predict whether a parallel single area garbage collector would be superior to a sequential single area garbage collector. Wadler [Wadler76] concludes that twice as much CPU time should be spent in a parallel garbage collector as in a sequential garbage collector. It is argued that this is not important since the garbage collection would be done with a parallel processor. There are still several questions about overhead that must be investigated before the parallel garbage collector can be seriously considered, however:

1) Should the parallel garbage collector processor be an inexpensive special purpose processor or should it be a general purpose processor?
2) If garbage collection is done with a general purpose processor, to what extent could this processor be used for executing other jobs rather than always garbage collecting?
3) One problem with multiprocessing systems is that each processor must have a large amount of high speed memory for its own use. To what extent will a parallel garbage collector share memory with the process running in parallel and to what extent will the garbage collector need memory of its own? How much overhead will this create for the running computation in added paging activity due to an inability to use all the high speed memory?
4) How much overhead does the parallel garbage collector require in normal

computation if the hardware has been carefully designed to perform as quickly as possible the more complicated store operation that is required by the parallel garbage collector?

5) Can the system and the hardware be designed so that there is no overhead for the normal computation unless a garbage collector is actually running in parallel with it?

Finally, whether to use parallel garbage collection depends upon two basic questions:

a) what is the total cost for parallel garbage collection vs. sequential garbage collection and

b) what interruption in service is associated with parallel garbage collection vs. sequential garbage collection.

At this time it appears that parallel garbage collection is more expensive but that sequential garbage collection requires more interruption of service. If question (5) above can be answered, "yes", then the system can be designed to support both parallel and sequential garbage collection and the user or programmer can select which will be used in each situation. This thesis does not consider the parallel garbage collector further and does not show how to combine it with the techniques presented in this thesis. This work is left to future research.

4.5 Inter-Area Links

One of the purposes of areas on ORSLA is to serve as the smallest piece of the system that can be garbage collected. Garbage collection of an area proceeds by copying all of the accessible objects in the area into a new copy of the area, and then deleting the old copy of the area. In order to allow an area to be garbage collected separately from the rest of the system, it is necessary to have a list of all the references stored outside of the area to objects in the area, i.e. a list of incoming references to the area. All the incoming references must be modified during garbage collection of the area so they will point into the new copy of the area. In addition, the incoming references can be used to define the set of immediately accessible objects in the area from which to start the garbage collection. It is also necessary, however, for an area to have a list of all the references stored in that area that point to

objects in other areas, i.e. a list of outgoing references from the area. Thus each inter-area reference is on two lists: the list of incoming references to the area the reference points into and the list of outgoing references from the area the reference is stored in. Before the old copy of an area is deleted, the outgoing references from the area must be removed from the lists of incoming references of the areas they point into. Thus each area must have a list of outgoing references and a list of incoming references. The elements of the lists are called inter-area links, each of which contains a single inter-area reference.

An example of an inter-area link is developed in Figure 10. Let us consider the objects x and y. Inside of object x is a reference to object y. This describes the state of affairs

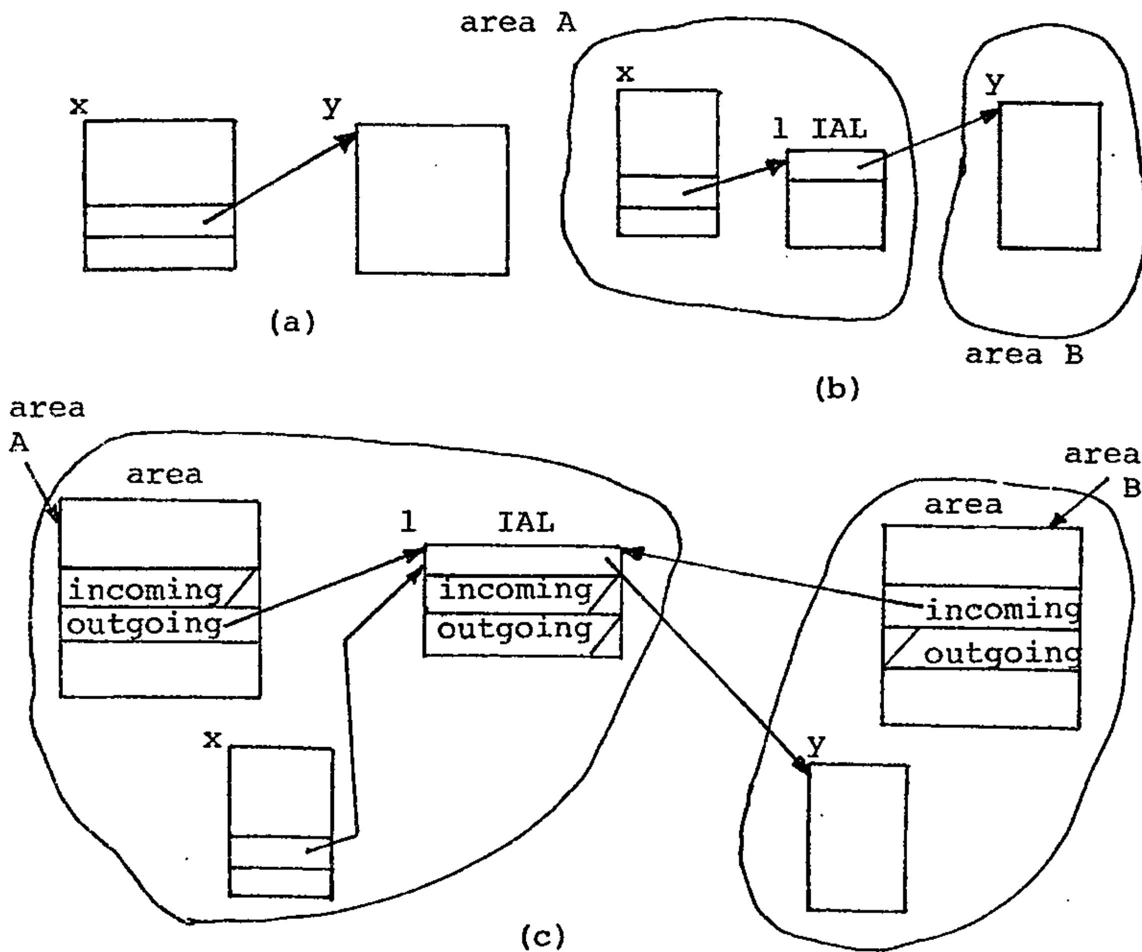


Fig. 10. Inter-Area Links

accurately if x and y are in the same area (see Figure 10a). If object x is in area A and object y is in area B, however, then the reference in x to y uses the inter-area link, l (see Figure 10b). Instead of storing the object reference to y directly in x, an object reference for l is stored in x; the object reference for y is in l.

It is necessary for the garbage collector to be able to find l given only a reference to area A or to area B. An object reference to an area points to an area object. Thus the area object must contain, among other things, an object reference to a list of incoming links and an object reference to a list of outgoing links. Each inter-area link, l, contains three object references as shown in Figure 10c: 1) a direct reference to the object (y) that l is pointing to, 2) a reference to the next inter-area link in the list of incoming links for the area (B) that l is pointing into, and 3) a reference to the next inter-area link in the list of outgoing links for the area (A) that l resides in. Thus each list of links has its root in the area object but is threaded through the inter-area links themselves. The list of incoming links to an area is threaded through the second object reference in the link, while the list of outgoing links from an area is threaded through the third object reference in the link. Figure 10c shows how each inter-area link is threaded onto these two lists. Since there is only one inter-area link in Figure 10c, the references in the link to the rest of the incoming and outgoing lists reference the *null* object. A more extensive example is shown in Figure 11, which contains an additional object w in area B and an additional object v in area A. Figure 11 also shows another area, C, that contains the object z. In addition to the inter-area link from x to y there are also inter-area links from x to w, x to z, y to v, and z to x. The lists of incoming links to area A starts in the area object for A and includes the link from y to v and the link from z to x. The list of outgoing links from A include the links from x to y, x to w, and x to z. The five links also form incoming and outgoing lists of links for areas B and C.

This chapter will only deal with the use of inter-area links; the problem of maintaining the lists of inter-area links will be covered in Chapter 5. In order to maintain the lists of inter-area links, however, it must be possible to find quickly what area an object is in. This is implemented by placing a reference to the area object into the page map on ORSLA. The importance of this feature at this point is that an inter-area link implicitly contains a reference to the area object of the area it resides in and a reference to the area object of the

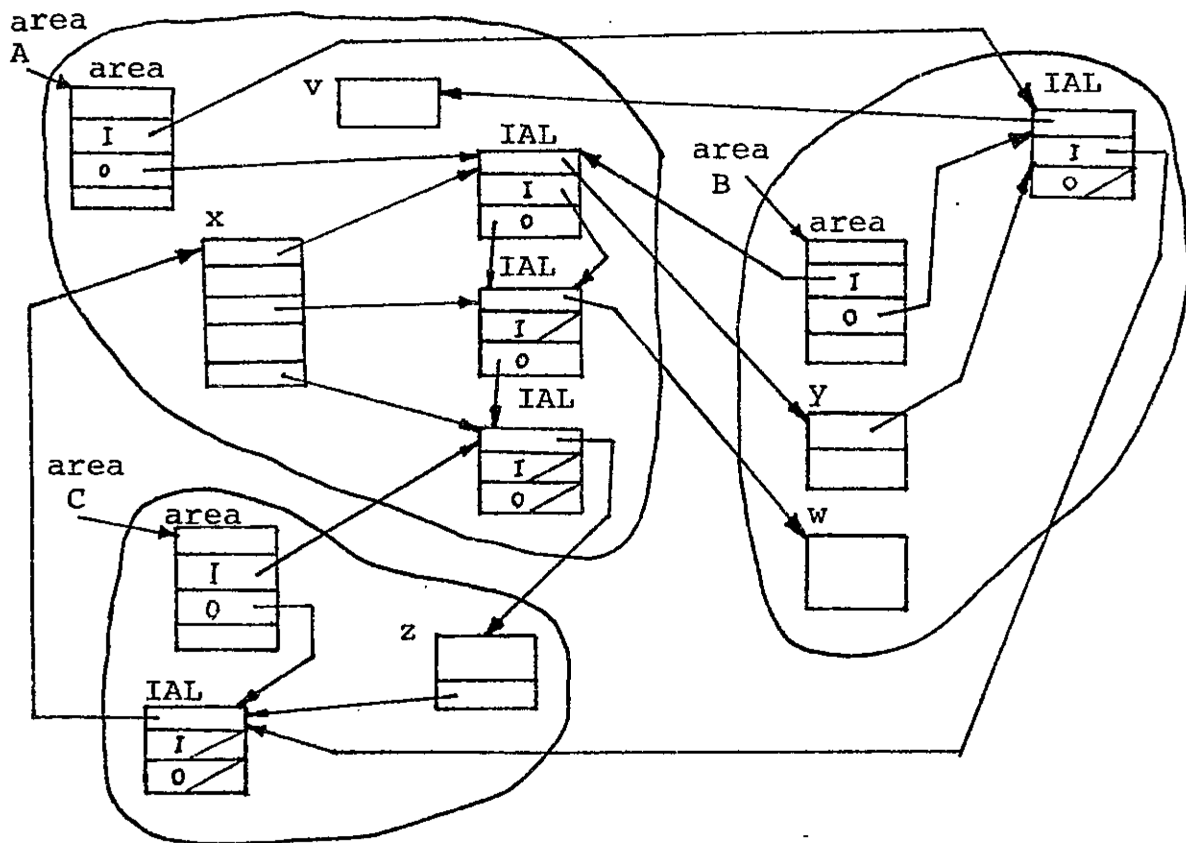


Fig. 11. Lists of Inter-area Links

---

area it is pointing into. These implicit references can be used to find the beginning of the list of incoming links and the beginning of the list of outgoing links given only a reference to an inter-area link. There is some question in the design of ORSLA about exactly how the lists of inter-area links should be implemented. An obvious alternative to the method presented here is for these lists to be doubly linked instead of being singly linked. The representation for the list of inter-area links presented in this thesis was selected because the size of an inter-area link is small and the cost of threading an inter-area link onto the beginning of a list of links is very low. The cost of unthreading an inter-area link, *l*, from a list of links may be high with this representation, however, since it is necessary to traverse the entire list from the beginning of the list to *l* and then to modify the previous link in the list. Future research on ORSLA may develop a superior representation for the lists of inter-area links, but the representation presented here is simple and is therefore superior for describing ORSLA.

An important design decision on ORSLA is in which area an inter-area link should be placed: the area it comes from, or the area it points into. An inter-area link must be placed in the area the link comes from in order to allow areas to be garbage collected separately. Figure 12 shows an improperly placed link. Area A in Figure 12 can be garbage collected correctly, but a problem arises when area B needs to be garbage collected. When area B is garbage collected, the inter-area link is able to identify *y* as an immediately accessible object in area B, but then the inter-area link itself is copied into the new copy of area B. It is then necessary to modify the reference in area A to the inter-area link, but there is no way of knowing where this reference is without garbage collecting area A. Thus ORSLA requires that an inter-area link be stored in the area the link comes from and not in the area it points into.

4.6 Cables

Areas on ORSLA can be divided into two very general classes: permanent areas containing long term data and local computation areas that contain the temporary results of computations. The information kept in permanent areas on ORSLA corresponds rather closely to the information kept in files on SAV and LAV systems, while the information

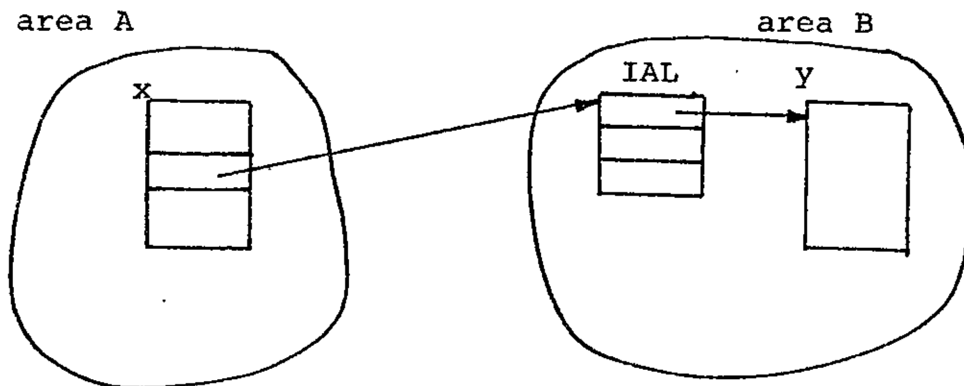


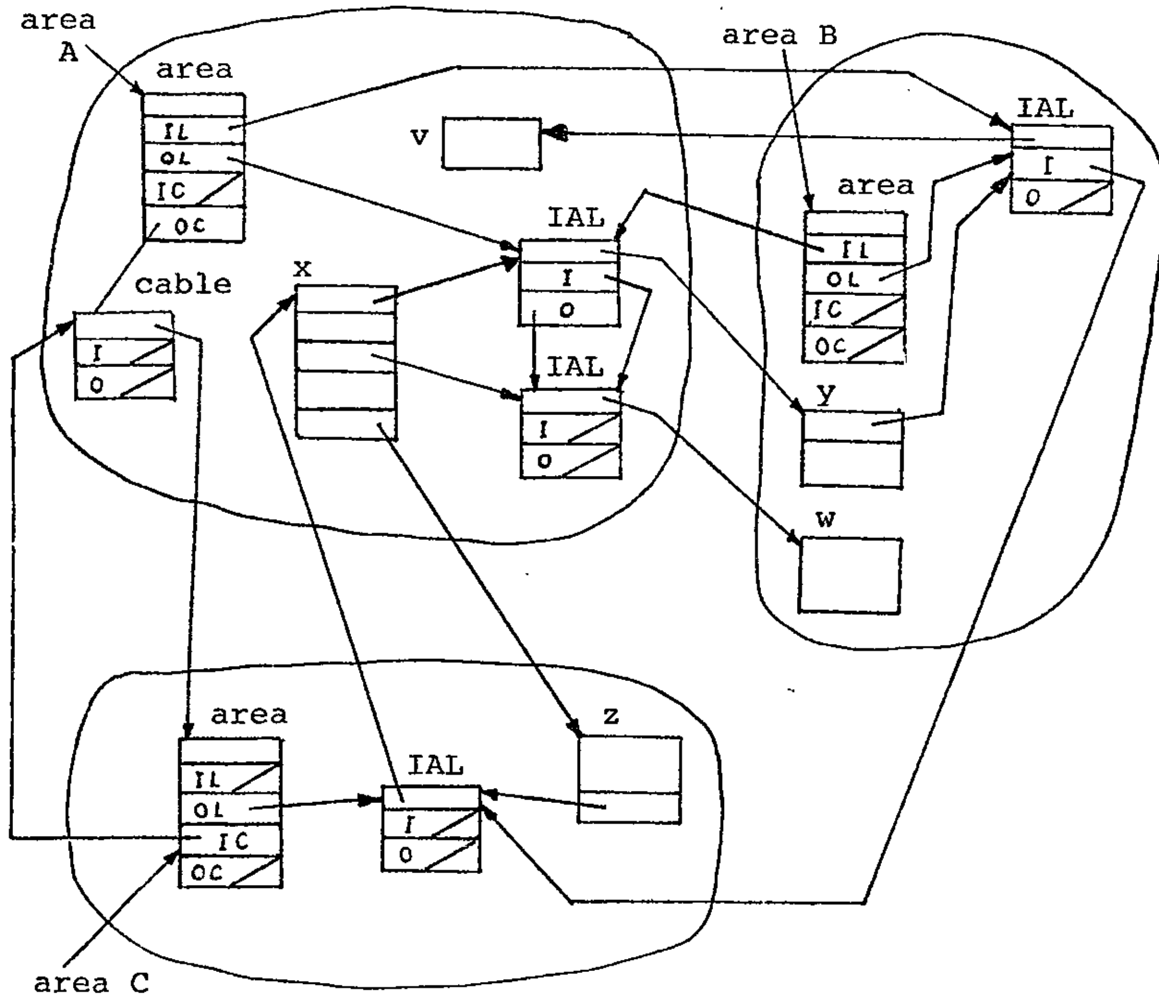
Fig. 12. The inter-area link should be in area A.

kept in a local computation area on ORSLA corresponds roughly to the information kept in the address space of a job on an SAV system, but corresponds more closely to the information kept in the segments in the process directory on Multics.

Inter-area links serve the needs of permanent areas rather well, but create too much overhead to be used with local computation areas. A local computation area will contain references to many different objects in permanent areas. Most of these references will only last for a short time, and so it would be expensive to construct an inter-area link for each of these references. Thus another mechanism, called a *cable*, is needed in addition to inter-area links. If there is a cable from area A to area C then object references stored in A to objects in C do not need inter-area links. A cable allows direct references. The area object has a list of outgoing cables and a list of incoming cables in addition to the lists of incoming and outgoing inter-area links. The existence of a cable from area A to area C is represented by a cable object on the list of outgoing cables from A and on the list of incoming cables to C. The cable object is stored in area A and contains a reference to area C and two other references, one for threading the list of incoming cables and one for threading the list of outgoing cables.

Figure 13 shows the example from Figure 11 except that a cable has been constructed from area A to area C. Thus, object x in area A contains a direct reference to object z in area C instead of using an inter-area link. This direct reference is allowed by the cable from area A to area C. The cable object is on the list of outgoing cables from area A and the list of incoming cables to area C.

A cable does not identify all references in the area it comes from, that is, it does not really provide a list of all references from that area. If there is a cable from area A to area C and if area C is being garbage collected, the references to C that are accessible from A can only be found by garbage collecting area A as well. Garbage collecting area A will of course find all the accessible object references within A, not just the references to objects in C. Area A, however, may be garbage collected by itself as long as there are no cables to area A. If area A generates garbage more quickly than area C and so needs to be garbage collected more frequently than area C, then the requirement that area A be garbage collected



Note that the direct reference from x to z is covered by a cable from area A to area C.
Fig. 13. Cables and inter-area links.

---

whenever area C is garbage collected is not too serious. If area A is a local computation area, this slightly enlarged garbage collection is preferable to the excessive overhead for creating inter-area links to all the objects in C that are used by the computation in A.

The vast majority of the cables on ORSLA go from local computation areas to permanent areas. In fact, local computation areas rarely use inter-area links at all. ORSLA

automatically constructs cables instead of inter-area links to handle references from local computation areas to permanent areas. Whenever a local-computation area (LCA) is garbage collected, the set of cables from the LCA is regenerated but only to those areas for which accessible object references still exist in the LCA. Since local computation areas are garbage collected frequently, they only have cables to those permanent areas that they are currently using or were using just a short time ago. Thus when a permanent area is garbage collected on ORSLA, it is only necessary to garbage collect the LCAs for the processes that were recently using the permanent area.

The cables in the system define a relation between areas. An area A is *cabled* to area B, written $A \mapsto B$, if there is a sequence of areas A, A_1, A_2, ... , A_n, B such that there is a cable from A to A_1, from A_1 to A_2, ... and from A_n to B. If area A is cabled to area B, ($A \mapsto B$), there may be object references in A that directly reference objects in B. The cabling relation is transitive and is formed by taking the transitive closure of the relation formed by all the individual cables. This gives us the important property of transitivity, i.e. if $A \mapsto B$ and $B \mapsto C$ then $A \mapsto C$ must be true. By this property, if $A \mapsto B$, then any object reference in B that does not use an inter-area link may be copied from B to A without making an inter-area link in A. The importance of this feature will be covered in more detail in Chapter 5.

Whenever an area is garbage collected, it is also necessary to garbage collect all of the areas that are *cabled* to it. Thus there is an incentive to keep chains of cables short. In fact, few cables will be used except for cables from local computation areas. An important goal of ORSLA is to be able to garbage collect each local computation area separately from all other areas on the system. This can be achieved if cables are not automatically generated from one local computation area to another, but are only generated from local computation areas to permanent areas. Thus chains of cables will never be generated automatically.

Once the mechanism of cables is available, however, it might as well be used wherever it is appropriate. Thus the user is able to explicitly create cables between areas. It is the user's responsibility in such cases to be sure that the increased size of some garbage collections will be worth the savings in inter-area links. Two problems arise when cables are

created explicitly by the user, however. When a cable is created from area A to area B, there may already exist some inter-area links from A to B that are now unnecessary. This cannot happen with an automatically generated cable because it is created when the first reference to an object in B is stored in area A. Unnecessary links are not a serious problem, but they do incur a small amount of unnecessary overhead that could be removed by converting the references to these inter-area links to direct references to the objects in area B. In order to do this, however, we must know the location of the references to the inter-area links; this can only be discovered by garbage collection. Thus a conversion of unnecessary inter-area links to direct references to cabled areas can only be performed during a garbage collection. As we will see shortly, the garbage collector must handle inter-area links very carefully; it is not convenient for the garbage collector to check the necessity of each inter-area link it finds. To help the garbage collector decide which inter-area links can be eliminated, there is an extra bit in each inter-area link called the *possibly_unnecessary* bit. When a user adds a cable from A to B, all of the outgoing links from A are checked and the *possibly_unnecessary* bit is turned on in those links that point to objects in B. The garbage collector will check the necessity of all inter-area links in which the *possibly_unnecessary* bit is on and will eliminate the unnecessary inter-area links.

The second problem with explicitly created cables is how to garbage collect them and how to delete them when they are no longer necessary. When an area is garbage collected, no special action is required to copy the needed cables that were generated automatically; they will be regenerated automatically in the new copy of the area by the same automatic mechanism that generated them in the old copy of the area. A cable from A to B will only be regenerated automatically in A' if a reference to an object in area B is copied into area A'. Explicitly created cables must be copied explicitly into the new area by the garbage collector, however, since they will not be regenerated automatically. Each cable contains an *explicit* bit to allow the garbage collector to distinguish automatically generated cables from explicitly created cables. Presumably an explicitly created cable from area A to area B should be destroyed when there are no references from objects in A to objects in B. This can be detected by marking explicitly created cables as being *unused* when they are first created. When an explicitly created cable is moved to A' during the garbage collection, it is also marked as being *unused*. If a direct reference is ever stored in A to an object in B, then

the *unused* mark will be removed from the cable. *Unused* cables behave as if they did not exist. The user may explicitly delete them from the lists of cables or may specify an option to the garbage collector that causes them to be deleted from the lists of cables if they are still *unused* at the end of a garbage collection.

Ordinarily, a cable from area A to area B cannot be simply removed from both the list of incoming cables to area B and the list of outgoing cables from area A. The system depends upon the existence of the cable on these lists if there are any accessible direct references from objects in A to objects in B. A cable may be deleted, however, if the direct references that depend upon the cable are modified to use inter-area links, but this can only be performed during a garbage collection of the area. Thus the operation of deleting a cable can be performed only during a garbage collection of the area it comes from. An explicit deletion of an explicitly created cable can be performed by merely turning off the *explicit* bit in the cable. The garbage collector will not copy the cable during the next garbage collection and will automatically create inter-area links for all the direct references that used the cable as the references are copied into the new area. Thus explicitly created cables may be explicitly deleted. Automatically generated cables can only be deleted by more sophisticated control of the automatic generation of cables, which will be discussed briefly in Chapter 7.

4.7 Local Computation Areas

The local computation area on ORSLA is a major architectural feature of the system; it contains the temporary storage for a single process. Thus the temporary storage for a single process is very compact, resulting in good locality of reference. All of the machine code and other permanent data on the system is kept in permanent areas and can be shared by all the processes on the system.

The most popular use of temporary storage is for procedure activation records that contain the local variables and compiler temporaries for a procedure. Each procedure activation record is an object on ORSLA. Usually activation records form a stack that allows the storage to be allocated, freed, and compacted at very high speed. Unfortunately the stack has a history of being one of the major sources of dangling references in programming

languages. On ORSLA, storage can be freed only by garbage collection or reference counts, since dangling references are not allowed on ORSLA. In order to achieve the high speed storage management that is possible with an activation record stack, reference counts must be used on the activation records. Reference counts were described briefly in Chapter 3; a detailed explanation of the reference count mechanism is beyond the scope of this thesis. I believe that it is possible to create a reference count scheme for activation records that allows the activation record stack to run as quickly as on any other system while at the same time allowing the retention of activation records when desired at a reasonable cost. Activation records should only be retained during debugging or backtracking, however. Co-routines and multi-processing should be handled by giving each process its own activation record stack. Thus the practicality of ORSLA depends upon the development of high speed automatic reference count algorithms for activation records. This is the most pressing problem left undone by this thesis.

An activation record stack requires its own pool of address space that is allocated and freed in its own way. Areas are used as pools of address space on ORSLA, but areas are used for many other things as well, such as keeping track of inter-area references. A stack operates as what I call a *subarea*, i.e., its only purpose is allocating and freeing storage, but it operates as part of an area on ORSLA and manages only a part of the address space assigned to the area. The activation record stack should be used in a rather limited way to allow freeing and compaction to continue at high speed. A process needs another subarea in which objects can be created that are not used in such restricted ways. Algol 68 has coined the term *heap* for this subarea. Thus each process has a local computation area associated with it that contains two subareas: an activation record stack and a heap. If the computation being performed in the local computation area involves co-routining or multi-processing, then the local computation area may contain several pairs of subareas containing an activation record stack and a heap; each pair of subareas may have a process associated with it. Most computations use only one process, however.

There are two interesting problems with areas on ORSLA that are somewhat alleviated by the local computation area. First, since ORSLA is a multi-user system, most areas are accessible to many parallel processes, some of which will be modifying information

associated with the area, such as the list of incoming links, the storage quota, or the free storage list. In order to coordinate these parallel processes, each area must have a lock that is set whenever the information associated with the area is modified or used. Thus it is necessary to set this lock when storage is allocated from the free storage list associated with an area. Allocating storage for activation records and other temporary objects must be a very high speed operation on ORSLA, however; it should be faster than setting a lock. The local computation area can alleviate this problem by providing each process with a pair of subareas that are used only by this one process; thus no locks need be set when allocating storage from these subareas.

The second problem with areas that is alleviated somewhat by the local computation area is the problem of which area to place an object into when it is created. The large number of areas on ORSLA makes this a difficult choice. Most objects, however, are intermediate results in a computation and will only have a temporary existence. Such objects should be placed into the heap in the local computation area. Note that LISP and Algol 68 do not allow objects to be placed elsewhere than in the heap. On ORSLA, when a programmer wants to create an object but does not specify in which area it is to be placed, the object is placed into the heap in the local computation area. This solution not only makes selection of the area in which to place an object much easier, but also encourages the use of the heap which provides higher speed allocation than other areas because it is not necessary to set a lock in order to allocate storage from the heap.

4.8 Using Inter-area Links in Machine Language Programs

We have now seen what inter-area links are and where they appear, but another important question is: how do they interact with machine language programs? ORSLA requires that inter-area links be invisible to machine language programs. Thus all machine code will be able to operate regardless of where inter-area links appear in the data. If inter-area links were not invisible to machine language programs, then a special instruction sequence would be needed to process an inter-area link. Although this instruction sequence could be made invisible to the assembly language programmer by the use of macros, the instructions would still appear in the machine language program and would require space to

store and CPU time to execute. A machine language program that did not perform this sequence whenever it dealt with an object reference would not be able to handle inter-area links in certain parts of the data. By making inter-area links invisible to machine code, we not only increase the generality of all the programs on the system, but also eliminate the need for the instruction sequence that processes inter-area links, thus significantly shortening and speeding up all machine language programs.

There are two ways to make inter-area links invisible to the machine language, both of which are used on ORSLA. One is the obvious brute-force method of treating a reference to an inter-area link as an indirect reference. Every instruction that is given an object reference for an inter-area link would behave as if it were using the object reference within the inter-area link instead. This method requires some complication in the CPU because every machine instruction must test for the existence of an inter-area link and process it correctly. Even then, however, this method is somewhat expensive because it is necessary for the CPU to access the first word of an inter-area link during each instruction that is given a reference to the inter-area link even if these instructions are performed in rapid succession. It would be more efficient if the first word of the inter-area link could be accessed once and the object reference obtained used several times. The main advantage of the brute-force method is that the direct reference to the object is so temporary and so internal to the CPU that it is not necessary for it to be covered by a cable.

The second method of making inter-area links invisible to the machine language is more efficient and less demanding on the CPU. The load instruction watches for object references that use inter-area links just as it watches for monitored locations. When an object reference for an inter-area link is to be loaded into a general register of the CPU, the direct reference to the object the link is pointing at is loaded into the register instead. From that point, the inter-area link no longer gets in the way of using this object. Further operations are not even aware that an inter-area link was ever used. Since this direct reference to the object remains in the general register for a long time, that is, longer than one instruction, it is necessary for this reference to be covered by a cable. The general registers are considered to be a part of the currently executing local computation area. The general registers can either be viewed as being part of the process object of the currently

executing process or part of the currently executing activation record depending upon where the registers are stored when an interrupt occurs. Both of these objects are in the local computation area. Thus the direct references loaded into general registers must be covered by cables from the local computation area. Thus this method of making links invisible may be used for objects in permanent areas since the local computation area may have cables to these areas, but the first method of making links invisible must be used for objects in other LCAs since the local computation area may not have cables to other LCAs.

4.9 The Area Object

We are now ready to consider most of the information that is kept in the area object. As the thesis progresses, however, we will occasionally find some other pieces of information that must be associated with an area. Appendix B lists all of the information kept in the area object. It is often difficult to tell exactly what information is kept in an object on ORSLA because users can define new representations for the object that keep additional information. The information within an area object is known, however, because the area is a sensitive object that is defined by the system. The user is given as much flexibility as possible, but since the entire system depends upon some of the information that is kept in an area object, the user is not allowed to define new representations for it. Some of the items kept in an area object are:

1) free list - This is a list of blocks of free storage and address space. When storage and address space are allocated from this area, the storage in this list decreases. The area may request that ORSLA allocate blocks of address space and storage to the area to increase the amount of storage in the area's free list.
2) address space quota - This is the maximum number of pages of address space that can be allocated to the area at any one time. The user is able to set the quotas on an area. The user can be sure of being able to use this much address space, but cannot get more without changing the quota.
3) address space used - This is the total number of pages of address space currently allocated to the area by ORSLA. This number may never exceed the quota.
4) pages allocated - This is a list of the pages of address space allocated to the area. When an area is deleted, e.g. after it has been garbage collected and all the accessible data

moved into a new copy of the area, all of these pages of address space are returned to ORSLA along with any pages of storage associated with these pages of address space. This list does not enable any of the address space to be accessed nor does it identify any objects that may reside in the address space.

5) storage quota - This is the maximum number of pages of physical storage that may be assigned to pages of address space that have been allocated to the area.

6) storage used - This is the number of pages of physical storage currently being used by the area.

7) incoming links - This is the list of inter-area links from other areas that are referencing objects in the current area. This list is used in garbage collection and is automatically maintained by ORSLA.

8) incoming cables - This is the list of cables from other areas to this area.

9) outgoing links - This is the list of inter-area links in the current area referencing objects in other areas. This list is also used in garbage collection and is automatically maintained by ORSLA.

10) outgoing cables - This is the list of cables from this area to other areas.

11) area information - This field contains several bits that specify information about the area. Two of the bits in this field are: the *LCA* bit to specify whether the area is a local computation area, and the *garbage_collecting?* bit to specify whether the area is being garbage collected. We will shortly see what these bits are used for and will also specify some more bits that are in this field.

12) miscellaneous information - This is a list of miscellaneous information that is associated with the area but is not sensitive information. The user of an area may keep information on this list and the garbage collector will keep some information on this list.

13) lock - Each area may be manipulated by any of the processes on the system. During manipulation of the area, it may be necessary for changes to be made in the free list or in the lists of incoming or outgoing links in this area. To prevent parallel processes from interfering with each other, this lock must be set whenever these lists are read or written so that only one process at a time will manipulate the lists.

We have now seen what information is kept in an area object. The information kept

within the area itself will correspond to the information kept within files on other systems. There is a more absolute way to characterize an area, however. An area on ORSLA is a module of storage. The mechanisms on ORSLA ensure that an area will be used in this way. First, related objects should be placed in the same area so they may be on the same page. Second, the number of inter-area links should be minimized to minimize the overhead for maintaining and using inter-area links. One way to do this is to have very large areas, but this solution defeats the entire purpose of areas: to break storage into pieces of reasonable size so it is possible to do garbage collection in a reasonable amount of time. Thus there are solid reasons on ORSLA for limiting the size of an area. Within this limitation, however, the number of inter-area links should be minimized. An area should represent a logical module[1] of related information, that may have many internal references, but will have relatively few external references.

The major factor that prevents an area from always being a single logical module of storage is that although the size of an area can vary widely, there are limits to the range of sizes of an area. An area cannot be smaller than a page of storage and even this size can be inefficient because it can cause internal fragmentation of more than 100% of the storage needed for the information in the area. Internal fragmentation occurs when the objects in an area do not use all of the storage in the pages of storage used by the area. To reduce the average storage wasted by internal fragmentation to about 10%, an area must have about 5 pages. If a single logical module of storage is smaller than 5 pages, it might be advantageous to combine it with another small logical module of storage, especially if the two modules are related and might be used together.

At the other end of the size spectrum, the size of an area is limited by the ability to garbage collect the area. Unfortunately, it is not known how much locality of reference there is in a copying garbage collector. In particular, the working set of the garbage collector may contain most of the pages in the area being garbage collected. Only a few of the pages in

---

1. It has been suggested [Constantine68] that a good modularization is one in which the number of interconnections between modules has been minimized.

the new copy of the area will be in the working set of the garbage collector, however. If there is not too much garbage in the area we should expect that the relative placement of many objects will remain the same, so the garbage collector may only need in its working set as few pages of the area being garbage collected as it needs from the new copy of the area. Thus the size of the working set of a copying garbage collector is uncertain, but it cannot be much larger than the size of the area being garbage collected. If it is assumed that a copying garbage collector does not exhibit locality of reference, then, given that it is unacceptable for the garbage collector to thrash, the size of areas should be limited to a little less than the size of high speed memory. But even if the garbage collector has excellent locality of reference, the size of an area is still limited by the amount of temporary storage available on the system in which to copy the area and for the other temporary storage used by the garbage collector. There should be enough disk storage available on the system for temporary use to allow the full use of high speed memory, so concern about the size of the working set of the garbage collector should be the limiting factor on the size of an area. Thus for each area, A, there will be a factor G_A that is related to the degree of locality of reference of the copying garbage collector when operating on the information in area A. Area A will become too large to garbage collect if A is larger than G_A times the size of high speed memory. Thus if a logical module of storage is too large, it must be broken into smaller pieces each of which is placed in its own area; of course, these areas must not be cabled to each other or it will be necessary to garbage collect them all at the same time, thus defeating the very purpose for which the separate areas were created.

4.10 Garbage Collection on ORSLA

Now that we understand what areas are and how the lists of inter-area links are represented we can ask how we are to garbage collect a single area separately from the other areas on the system. Probably the only areas on ORSLA that can truly be garbage collected by themselves are local computation areas; a permanent area must be garbage collected together with the local computation areas of the processes that have been using the permanent area recently. By garbage collecting only a few areas at one time, we reduce the amount of real time taken by a garbage collection and reduce the working set of the garbage collector. We also allow garbage collections to occur at different frequencies in different

areas so that garbage collection effort can be concentrated on those areas that need it. Another advantage of garbage collecting only a few areas is that it is only necessary to stop the processes that are using information in those areas. Processes that are not using any of the information in the areas being garbage collected will not be cabled to those areas and so may proceed with their computations regardless of whether those areas are being garbage collected or not.

If processes are able to execute in areas that are not being garbage collected, it becomes natural to ask whether these processes may suddenly begin another garbage collection that involves a different set of areas from those already being garbage collectd. It is obviously desirable to allow multiple simultaneous garbage collections, but the garbage collector must be carefully designed to allow them to occur. Problems arise because the garbage collector looks at all of the incoming links to the areas being garbage collected. Normally, whenever a process looks at an object in an area, a cable is constructed from the LCA running the process to the area. Thus we might expect the garbage collector to require cables from the LCA running the garbage collector to all the areas being garbage collected and all the areas that have inter-area links into these areas. But then none of these areas could be involved in another simultaneous garbage collection since the LCA running our garbage collector would also have to be part of the other garbage collection. This problem can be avoided by carefully designing the garbage collector so it does not require cables from the LCA to areas that are linked via inter-area links to the areas being garbage collected. Then areas that are only connected via inter-area links could be involved in separate simultaneous garbage collections.

Garbage collection is performed by a procedure that takes an area, A, as an argument. Area A and all the areas cabled to A must be garbage collected together. Garbage collection of just a few areas requires more phases than those mentioned in section 4.1. The first phase finds all the areas cabled to A, makes sure they are not part of another garbage collection, and reserves them for this garbage collection. In addition, the first phase prevents the construction of new cables to these areas and stops the processes executing in these areas. The set S is the set of areas being garbage collected. S is initialized to {A}. For each area, X in S, area X is locked and the *garbage_collecting?* bit in the *area_information* field of X is

inspected. If this bit is zero, then it is set to one and all of the areas on the list of incoming cables to X are added to *S* without creating duplications. If X is an LCA, then all of the processes executing in X are halted. A process is executing in X if X is the local computation area for the process. A process is stopped for the garbage collection even if it is asleep waiting for an event to happen. All of the processes that have been stopped in X are placed on a list, P, that is put in the miscellaneous information of area X. The fact that the *garbage_collecting?* bit in X has been set will cause any processes that attempt to construct a cable to area X to be halted and placed on the list, P, in the miscellaneous information in area X. At this point, area X is placed on the list, C, of areas that have been claimed for this garbage collection. Finally, area X is unlocked. If, when the *garbage_collecting?* bit is inspected, it is already one, then it is not possible to perform this garbage collection at this time because area X is already involved in another garbage collection. Area X is unlocked, the *garbage_collecting?* bits in all of the areas on the list C are turned off, all of the processes on the list, P, of processes that have been stopped for the garbage collection are resumed, and the garbage collection is abandoned for a later time when all of the areas cabled to A will be available.

When the first phase has been completed successfully on all of the areas in *S*, *S* will contain all of the areas cabled to A, which will have been found to be available for garbage collection and will have been reserved for this garbage collection. This entire phase of the garbage collector must be performed by special system code that does not generate cables to the areas in *S* even though the area objects in these areas are being used. No cables should be constructed to these areas until after it is known that the garbage collection will actually occur.

Once it has been determined that the garbage collection will actually occur, it is possible to begin performing operations that cannot be reversed. The second phase of the garbage collector prepares each of the areas in *S* for garbage collection. For each area, X, in *S*, a cable is constructed to X from the local computation area in which the garbage collector is running. A new copy of area X, area X', is constructed and is added to the set *S'* of new areas produced by the garbage collector. A cable is constructed from the LCA running the garbage collector to area X'. All of the accessible information in area X will eventually be

copied into area X' by the garbage collector. The initial size of area X' is a function of the estimated storage used by accessible objects in area X. As we will see later, the page map on ORSLA contains a *GC* bit. The *GC* bit for all of the pages in X are now turned on. It is now very easy to determine that area X is being garbage collected. A reference to the LCA running the garbage collector is now placed in the miscellaneous information of area X so it is easy to determine which garbage collection X is part of. Although most of the processes that may access objects in X have been stopped because their LCAs are cabled to X, some processes may use objects in X via inter-area links. Thus all the incoming links to area X are inspected and if any come from LCAs, the processes in these LCAs are stopped and placed on the list, P. In addition, if any process attempts to construct any additional inter-area links to area X, this process must be stopped and placed on the list, P, in the miscellaneious information in area X. Other garbage collections, however, must be allowed to construct copies of existing links. Garbage collection is carefully written so it will merely copy a link and will not access the object. The cables from X to other areas are now checked to see if any have the *explicit* bit turned on. Any *explicit* cables from X must be added to X'. If a cable from X is to area Y in *S*, then the cable from X' should go to the corresponding area Y' in *S'* instead of to area Y. These cables are marked as being *explicit* and *unused*. When a direct reference uses one of these cables, the *unused* marking will be removed. Note that although I have only been concerned about outgoing cables, all of the incoming cables are transferred to the areas in *S'* as well because all of the incoming cables to areas in *S* are outgoing cables from other areas in *S*. Finally, an external mark data base (M_X) for area X must be created in the LCA. Each area has its own mark data base in order to improve locality of reference to mark data bases. References to X' and M_X are placed in the miscellaneous information of area X.

The copy phase consists of processing the areas in S one at a time beginning with area A and calling the *collect* procedure on each of the immediately accessible objects in each area X in *S*. Since the garbage collection cannot identify which objects outside of the areas in *S* are accessible, it must assume that all of the objects outside of *S* are accessible. Thus the first source of immediately accessible objects in area X in *S* is the incoming links from areas outside of *S*. These are found by scanning the list of incoming links to X and checking each to see whether it comes from another area in *S*. An area is a member of *S* if the

garbage_collecting? bit is on in the area and if the portion of the miscellaneous information in the area that contains a reference to the LCA running the garbage collection it is part of is pointing at the current LCA. If so, it is marked as *possibly_unnecessary* so that the object it points to will be marked from only if the link is found to be accessible during garbage collection of the area that the link comes from. If an incoming link comes from an area outside of *S*, however, then the *collect* procedure is called on the object pointed to by the link. The *collect* procedure returns a reference to the new copy of this object in area X' and the link is modified to point to this object. The incoming link must be removed from the list of incoming links to area X and added to the list of incoming links to area X'. After the list of incoming links has been processed, then the list, P, of processes stopped in this area is processed. All of the objects on this list that reside in *S* are *collect*ed and the new copies of these objects are placed on the list P in area X in place of the old copies. Finally, the activation record for the procedure that called the garbage collector is an accessible object. This last object need not be considered if the LCA running the garbage collector was created solely for performing the garbage collection because the process running the garbage collector will be destroyed after the garbage collection. The *collect* procedure is called on the activation record of the procedure that called the garbage collector if it resides in area X. This completes the copy phase for area X. Each of the areas in S is processed in this way beginning with area A, the main area being garbage collected.

The *collect* procedure is very similar to that described in section 4.2. There are some important differences, however. It is necessary for an object in area X to be copied into its corresponding area, X', and to be registered as marked in M_X. In addition, the mark data base M_X must be used to determine whether objects in area X have been marked. The garbage collector maintains a pair of variables, N and M, that specify the new area that objects are being copied into and the mark data base for the area currently being processed. These variables remain set as long as the garbage collector operates on intra-area references, but new values for these variables must be found whenever the garbage collector encounters an inter-area reference. Many inter-area references are direct references that are covered by cables. If a direct inter-area reference points into an area outside of *S*, however, the *collect* procedure should not be called on this object, rather the reference to the object should merely be copied into area X'. Thus the garbage collector must detect when a direct

reference is an intra-area reference and, if it is an inter-area reference, the garbage collector must determine whether it is pointing into an area in *S*. Fortunately, this can be determined by simply inspecting the *GC* bit in the page map for the page pointed into by the inter-area reference. If the *GC* bit is on, the area is in *S*. This is true because no area, Z, to which an area, X, in *S* is cabled can be involved in another garbage collection. If Z were involved in a garbage collection, all the areas cabled to Z would also be involved in the same garbage collection, so area X would be involved in the same garbage collection as area Z.

The new modified Algol program needed for *collect* is shown in Figure 14. The *collect* procedure is now an internal procedure of the *collect_in_new_area* procedure which merely sets up the variables *N* and *M*. The argument and value of the *collect_in_new_area* procedure are the same as for the *collect* procedure. The *collect_in_new_area* procedure looks in the miscellaneous information of the area that *y* is in (A), to find the corresponding area (*N*) that objects in A are to be copied into and also to find the mark data base (*M*) that is to be used for objects in A. Then the *collect_in_new_area* procedure just calls the *collect* procedure. The *externally_marked?*, *new_copy*, and *externally_mark* procedures now take the mark data base they operate with as an argument. Similarly the *allocate_new_copy* and *create_new_link* procedures take the area to allocate in as an argument.

The most significant change in the *collect* procedure from the version used when the entire address space is garbage collected at once occurs when an object, *x*, is marked from and all its information is copied into *new_x*. This section of the *collect* procedure must be prepared to handle five different mechanisms. First, it must be prepared to handle storage monitors, but storage monitors are handled just as they were earlier. The *GC_load* operation returns an object reference whose *storage_monitor* bit may be on. If so, the bit is turned off with the *deactivate_monitor* operation and when it is finally determined what object reference should be stored in the new copy of the object, the storage monitor is reactivated in the new object. Once storage monitors have been taken care of, there are four different kinds of object references that the garbage collector must handle. The simplest kind of object reference is for atomic data. Object references that do not contain addresses can be simply copied into *new_x*. Non-atomic object references that do contain addresses, however, must be handled more carefully. A direct intra-area reference, i.e. *area*(*w*) = *area*(*x*), is

```
procedure collect_in_new_area(y);
    begin object y, M; area A, N;
    procedure collect(x);
        begin object x, new_x; integer i;
            if externally_marked?(x, M) then return new_copy(x, M);
            new_x := allocate_new_copy(x, N);
            externally_mark(x, new_x, M);
            for i := 1 to size(x) do
                begin object z;
                    GC_load(z, x[i]);
                    begin object w;
                        if storage_monitor?(z) then w := deactivate_monitor(z);
                                    else w := z;
                        if ~atomic?(w) then
                          if IAL?(w) then w := create_new_link(w, N);
                          else if area(w) = area(x)
                              then w := collect(w);
                              else if garbage_collecting?(area(w))
                                    then w := collect_in_new_area(w);
                                    else ;
                        if storage_monitor?(z) then set_storage_monitor(new_x[i], w)
                              else new_x[i] := w;
                    end
                end
            return new_x;
        end
    A := area(y);
    N := corresponding_area(A);
    M := mark_data_base(A);
    return collect(y);
end
```

Fig. 14. COLLECT_IN_NEW_AREA Procedure

---

processed by calling the *collect* procedure recursively so the current values for N and M will continue to be used for w. Since the page map on ORSLA contains an object reference for the area containing the page, the test ($area(w) = area(x)$) can be performed easily. If, on the other hand, a direct reference is an inter-area reference, then we must determine whether it is pointing into an area in S. If $area(w)$ is not being garbage collected, then it is clearly not a member of S, and so the object reference should be used as it is. If $area(w)$ is being garbage collected, however, then we have seen that this means that it is a member of S, so the new

version of the object must be found by calling the *collect_in_new_area* procedure which finds new values for N and M before calling the *collect* procedure for w. The only alternative to a direct reference is a reference that uses an inter-area link. It is much more difficult to determine whether inter-area links rather than direct references are pointing into an area in S because if the *GC* bit is on in the page pointed into by the link, it may merely indicate that the area is involved in another garbage collection. Furthermore, inter-area links must be handled carefully to prevent the construction of a cable from the LCA running the garbage collector to all the areas to which there are inter-area links from areas in S. Thus the *GC_load* operation in the *collect* procedure returns the reference to an inter-area link; it does not automatically obtain the direct reference to the object pointed to by the link as the ordinary load instruction does. The *collect* procedure checks the type code on the object reference returned by *GC_load* to determine if it uses an inter-area link. If an inter-area link is found, it is assumed that the link points into an area outside of S, so the inter-area link itself is copied with the *create_new_link* operation but the *collect* procedure is not called on the object pointed to by the link. The *GC_load* operation does obtain the direct reference to the object pointed to by an inter-area link in which the *possibly_unnecessary* bit is on, however. Thus the *collect* procedure treats *possibly_unnecessary* inter-area links as direct inter-area references. This feature of the *possibly_unnecessary* bit, combined with setting the *possibly_unnecessary* bit in incoming inter-area links from other areas in S, act together to prevent any problems from arising from the invalid assumption that all inter-area links from areas in S are to areas outside of S.

When all of the initially accessible objects have been processed and the incoming inter-area links modified to reference the new copy, all of the information has been copied into the new areas. There are no inter-area links into the areas in S from outside the areas in S because they have all been modified to reference objects in areas in S'. There are no cables to the areas in S that come from areas outside of S because if there were, the area they come from would have been included in S as well. The processes that will be resumed after the garbage collection now all reside in areas in S'. There are no references to any of the old objects anywhere in the system. We may now free the storage and address space for all of the areas in S, resume all the processes that were stopped for the garbage collection and return to the new copy of the activation record for the procedure that called the

garbage collector.

We have now seen how the garbage collector works, but it is not clear how the various strange ways of handling inter-area links interact. For example, why do we set the *possibly_unnecessary* bit in incoming links from other areas in S, and why do we assume that outgoing inter-area links from an area point into areas outside of S when we know that there may be inter-area links between areas in S? The reason for handling inter-area links so differently from direct references is to avoid constructing cables from the LCA that is running the garbage collector to all the areas that are connected by inter-area links to the areas being garbage collected. It is this requirement that forces the *GC_load* instruction to return the object reference for an inter-area link when one is encountered rather than returning the direct reference to the object the link points to, as the normal load instruction does. The garbage collector assumes that outgoing inter-area links point into areas outside of S so that only the outgoing link itself will be copied when it is found to be accessible; no direct reference to the object linked to will exist in the LCA that is running the garbage collector. The incoming inter-area links from areas outside of S are assumed to be accessible, so they are marked from when the area they point into is garbage collected. Since the garbage collector will determine which objects in areas in S are accessible, however, the incoming links from other areas in S are not marked from when the list of incoming links for an area is processed. Another reason for not marking from the incoming inter-area links from areas in S is that these links should not be modified to point to the corresponding objects in areas in S' because none of the object references in areas in S are modified by the garbage collector to point to objects in S'. These techniques work well for handling both incoming and outgoing links that are connected to areas outside of S, but how can we be sure that inter-area links between areas in S will be marked from when they are found to be accessible? An accessible inter-area link from area A to area B, both of which are in S, will eventually be copied into area A' in S'. If the list of incoming links of area B is processed after this link has been copied, then since the new copy of the link is from an area in S' rather than being from an area in S, the new copy of the link will be marked from. If, on the other hand, the link from A to B is found to be accessible after the list of incoming links in area B has been processed by the garbage collector, then the link will be found to be *possibly_unnecessary* and so the object pointed to by the link will be marked when the link is

found to be accessible. When the reference to the new copy of the object pointed to by the link is stored in an area in S', then an inter-area link will be created automatically if one is needed. Thus, when the garbage collector processes the list of incoming links to an area B in s, it processes links from other areas in S that have already been found to be accessible by modifying the copies of these links that now reside in areas in S'. The garbage collector also marks all incoming links to B from other areas in S as *possibly_unnecessary* so that if they are found to be accessible after B has been processed, the objects pointed to by the accessible links will be marked when the links are found to be accessible. Once the list of incoming links to B has been processed, none of the links from areas in S to objects in B will be copied into areas in S'. Instead, links to the new objects in area B' will be created in areas in S'. Thus the list of incoming links to area B need be processed only once because no new inter-area links to B will be created after B has been garbage collected.

4.11 Multiple Simultaneous Garbage Collections

The major goal of garbage collection of a single area is to reduce the amount of work needed to collect the garbage on the system by allowing the garbage collector to be concentrated where it is needed rather than forcing it to uselessly garbage collect areas in which no garbage has been generated since the last garbage collection. A secondary goal of the garbage collector on ORSLA, however, is to allow a single process to garbage collect its LCA without interfering with computation or garbage collection elsewhere in the system. Although this is achieved fairly well, there are some important points in the garbage collector that improve this performance and there are some important ways in which this goal is not achieved. When an area is garbage collected, so are all the LCAs that are cabled to this area. Once a garbage collection has begun, however, none of the other areas that these LCAs are cabled to may be garbage collected until this garbage collection is over. Inter-area links between LCAs reduce the size of the garbage collection by allowing these LCAs to be garbage collected separately, but it is necessary to halt execution in all LCAs that have links to LCAs that are being garbage collected. Although normal processes cannot execute in LCAs that have links to LCAs that are being garbage collected, it should be possible for another garbage collection to run in these areas.

The most severe problem of multiple simultaneous garbage collections is a deadlock. The LCA running a garbage collector cannot be cabled to an area involved in another garbage collection. Thus the garbage collector must not add cables to new areas once the garbage collection has begun. Furthermore, the garbage collector must assemble the areas that will be garbage collected, make sure that none of these areas are already involved in another garbage collection, and reserve these areas for the current garbage collection before actually beginning the garbage collection. The task of reserving the areas for the garbage collection cannot result in deadlock if, before the entire set of areas has been reserved, it is possible for the garbage collector to abandon the garbage collection and release the areas it has already reserved if it finds that an area needed for the garbage collection is already involved in another garbage collection. Once the set S of areas that will be garbage collected has been reserved, however, another garbage collection will not be able to garbage collect any of the areas in S. Another garbage collection will have to garbage collect one of the areas in S if it tries to garbage collect any of the areas, T, that areas in S are cabled to. Thus another garbage collection must avoid not only the areas in S, but also all the areas in T. Since areas are usually considered to be cabled to themselves, the set T, containing all of the areas that areas in S are cabled to, contains S as a subset. Thus the areas in T specify the part of the system that cannot be used in any other simultaneous garbage collection. If the set T were to grow during the garbage collection, however, deadlock might occur when garbage collectors were halted because they were creating cables to areas involved in another garbage collection. Thus the prevention of deadlock due to the garbage collector is performed in two parts: reserving the set S during a phase of the garbage collector that can be undone or abandoned, and writing the garbage collector so that no new cables are added from the LCA running the garbage collector to areas outside of T.

The requirement that the LCA running the garbage collector not have cables to areas outside of T makes the processing of the incoming and outgoing inter-area links somewhat difficult because the lists of incoming links to the areas in S are threaded through objects most of which reside in areas outside of T. On the other hand, there is the serious question of whether a subsystem programmer should be able to write the code that searches, modifies, and rethreads these lists. Since system reliability depends upon the correctness of these lists, the manipulations of the incoming and outgoing inter-area links should only be performed

by sensitive system code. This sensitive code, however, can have direct references to the incoming inter-area links and the area objects for areas outside of T without having these direct references covered by cables. Thus the part of the garbage collector that processes the list of incoming inter-area links and invokes the *collect_in_new_area* procedure on the objects pointed to by inter-area links from areas outside of S must be written as sensitive system code. Similarly, the handling of outgoing inter-area links must be handled carefully. The *collect* procedure that has been presented does not cause direct references to be created in the LCA to objects in areas outside of T that are linked to from areas in S. The only operation performed by the *collect* procedure that must operate on objects in areas ouside of T is the *create_new_link* operation which must thread an inter-area link onto the list of incoming links for an area outside of T. Thus the *create_new_link* operation must be provided by sensitive system code that does not cause a cable to be constructed to the area the link is pointing into. This operation must already be sensitive system code, however, since the link must be threaded correctly or system reliability will be threatened. Although the need for parts of the garbage collector to contain sensitive system code limits somewhat the ability of a user to write a new kind of garbage collector, this limitation should be kept as small as possible by designing the interface between the sensitive system code and the garbage collector code so that the garbage collector can control the garbage collection and the sensitive code merely prevents construction of cables that are not really necessary and ensures that the lists of inter-area links are always correct. The detailed specification of this interface will be left to the system implementor, however.

Multiple simultaneous garbage collections are possible because there are almost no interactions between the garbage collections. One important interaction was just considered that could lead to deadlock if the garbage collector were not carefully designed. Another interaction occurs, however, when there is an inter-area link, l, from area A which is involved in one garbage collection to object x in area B which is involved in another garbage collection. At some point in the garbage collection of area A, the *create_new_link* operation will be performed on link l. The *create_new_link* operation creates the inter-area link l' in area A' to the same object pointed to by l. If l has already been processed by the garbage collection in area B, then l' will initially be created to point to object x' in B'. If, on the other hand, l has not been processed by the garbage collection in area B, then l' will

initially be created to point to object x in area B. Since this incoming link has not been processed, however, the garbage collection in area B has not finished processing the incoming links to area B. It should thus be able to handle an additional incoming link without much difficulty. The garbage collection in area B will process link l' in the same way it processes link l, causing them both to point to x' in area B' by the time the garbage collection in area B is complete. Thus regardless of the order in which these garbage collections proceed, inter-area links between them will be processed correctly as long as the *create_new_link* operation is made an indivisible operation by the use of locks, as long as modification of incoming links is also made indivisible, and as long as the garbage collector will correctly handle additional incoming links that have been constructed to an area after the garbage collector has begun processing the list of incoming links to the area but before the garbage collector has completed processing the list of incoming links. Once the list of incoming links to an area has been completely processed, no new incoming links will be constructed to this area, rather links will be constructed to the new copy of the area instead.

4.12 Multiple-Area Cycles

The algorithm presented in this chapter does a perfect job of garbage collection if all the references from other areas are in fact accessible within the areas they come from. If this is not true, however, then it is possible that some of the objects that were considered to be accessible are not, in fact, accessible and should have been destroyed. If these inaccessible inter-area links form a tree, then when the area containing the root of the tree is garbage collected some inter-area links will disappear and eventually all of these inaccessible objects will be destroyed. If these inaccessible inter-area links form a cycle, however, it appears that these inaccessible objects will never be destroyed.

For example, consider Figure 15a. Object x in area A has a reference to object y in area B, but there are no references to object x elsewhere in the system. When area B is garbage collected, y is considered to be accessible because l is on the list of incoming links to B. When area A is garbage collected, however, x is not found to be accessible and so x and l are destroyed. Then area B can be garbage collected and y will also be destroyed. If there is a link from y to x, however, as in Figure 15b, then when area A is garbage collected, x is

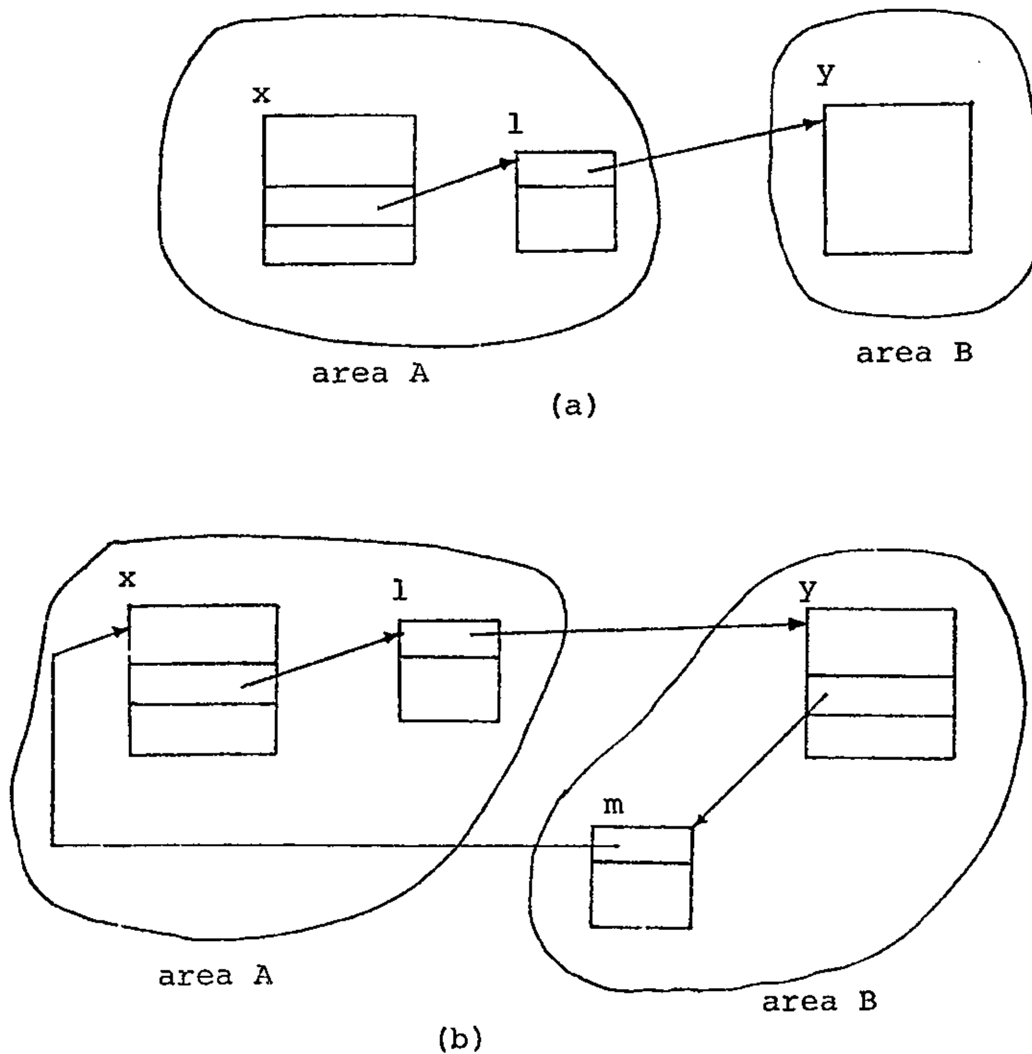


Fig. 15. An Inter-area Cycle

---

considered to be accessible because *m* is on A's list of incoming links. Since an inter-area cycle is formed, the techniques presented so far will not reclaim the storage in objects in the cycle or in objects referenced from objects in the cycle.

The main problem is that the incoming inter-area links do not necessarily point to accessible objects, thus some other, more reliable method of finding the immediately accessible objects is needed. On SAV and LAV systems, the user usually has a pretty good

idea of what information is kept in each file. The user is able to manipulate the information in the file by using the name of the file. A similar mechanism will be needed on ORSLA to provide the user with names of objects that the user will manipulate explicitly with user commands. The user will be able to type in the name of an object and the file system will convert it to an object reference which can then be used to manipulate the object. Any object that is named by the file system is clearly accessible, however. Furthermore, names on any objects in an area should describe all the information in the area and this information should be found by tracing from the named objects through other objects in the area by way of intra-area references. Thus the objects named by the file system should allow the garbage collector to find all the accessible objects on the system except for those objects that are intermediate results of active computations. The copy phase for an area could then be broken into two parts. The first part of the copy phase would last while the objects in the area named by the file system and the active processes in the area are used as immediately accessible objects. After the first part of the copy phase, all of the accessible objects in the area should have been copied into the new copy of the area. During the second part of the copy phase the incoming inter-area links are used as immediately accessible objects. Any unmarked objects in the area that are pointed to by incoming inter-area links, however, do not appear to belong in this area. Two important benefits would be gained if the garbage collector would move such objects into the area from which they are being referenced rather than the new copy of the area being garbage collected. First, the garbage collector would further ensure that areas are modules of related information and thus have good locality of reference and an acceptable number of inter-area links. Second, as we see by inspecting Figure 15b again, the garbage collector would either move x into area B or y into area A depending upon which area was garbage collected first. What was originally an inaccessible inter-area cycle has now become an inaccessible intra-area cycle whose storage will be reclaimed as soon as the area in which it resides is garbage collected. The automatic movement of objects from one area to another performed by this modification to the garbage collector is considered to be the result of an *automatic mover* that is implemented by the garbage collector. The automatic mover consolidates any arbitrarily complex, inaccessible multiple-area cycle into a single area where it will be reclaimed. Unfortunately, the automatic mover raises some serious protection issues. Solutions to these problems are proposed in Chapter 6, where the automatic mover is

presented and analyzed in detail.

4.13 The Effect of Errors in the Garbage Collector

A classical problem with garbage collectors has been the seriousness of errors in the garbage collector. A compacting garbage collector, especially, is able to destroy all the objects in the address space being garbage collected even when the bug it contains is relatively minor. The garbage collector on ORSLA is less likely to cause such disasters. The most important reason for this is that the garbage collector on ORSLA does not need to violate the restrictions on the use of object references, so ORSLA can continue to enforce these restrictions within the garbage collector. A copying garbage collector does not scan all storage, rather it only scans objects for which it has found object references. Once the handling of the external mark data base is correct, there is a real possibility on ORSLA of allowing the *collect* operation to be defined separately for each data type as a part of the data type definition. If the code is grossly incorrect, it may be disastrous for objects of that type, but enforcement of the restrictions on the use of object references will prevent the disaster from spreading to other data types; furthermore, bugs will be limited to the objects in which they occur. The ability to write a special *collect* procedure for each new data type would greatly increase the range of possible behaviors for new data types.

4.14 Other Related Work

The most distinctive feature of ORSLA is that areas are used to group objects together in a linear address space to achieve locality of reference and also to allow garbage collection of small parts of the address space. Three other researchers seem to have found related mechanisms, but on closer inspection, it appears that this related work is either different from ORSLA or has been extended by ORSLA.

4.14.1 AED Free Storage Package

The oldest related work is the AED free storage package [Ross67]. Free storage is broken into *zones* which have more in common with the areas on ORSLA than do areas in PL/I. The remarkable feature of zones is that a zone, like an area on ORSLA, is not a

single contiguous block of storage, but may have an arbitrary number of blocks of storage. In addition, the pointer to a zone points to a *plex* (similar to an object on ORSLA) that describes the zone (similar to the area object on ORSLA). Finally, different zones can be managed in different ways. The concept of the *help procedures* in zones is similar to the definition of a data type on ORSLA and so is consistent with the philosophy of ORSLA. This part of the design of areas is beyond the scope of this thesis, however.

There are several important differences between zones and the areas on ORSLA, however. First, zones were designed to manage the available free storage within the address space of an SAV system. Ross says, "Although the present system applies only to the control of uniform core storage, the zone technique provides the proper basic framework for efficient use of ... massive backing stores."[1] I agree with this statement, but I suggest that some mechanisms need to be added to zones to handle massive backing stores. In particular, zones have no special mechanisms for handling pointers stored within a zone that point to plexes in other zones. Second, Ross says that the automatic reclamation of storage is inefficient whether it is done by garbage collection or reference counts[2]. Implicit in the design of ORSLA, however, is the idea that automatic reclamation of storage is not inefficient and furthermore, that the dangling references that can appear in AED are unacceptable. This fundamental difference in approach is where ORSLA diverges from Ross' philosophy.

4.14.2 Greenblatt's LISP Machine

The LISP machine [Greenblatt74, Knight74] being built at MIT also uses areas. The programmer must specify in which area each object is to be placed, so the areas on the LISP machine help increase locality of reference. As on ORSLA, the 2^{23} word address space on the LISP machine is divided into pages; an area must contain a whole number of pages. Also, areas on the LISP machine have information associated with them such as a free list

1. [Ross67], p. 482.
2. [Ross67], p. 485.

and a list of outgoing references (called an *exit vector*). The LISP machine does not continually maintain the validity of the exit vector, however: it is regenerated whenever the area is garbage collected. Thus, if an area has not been modified since the last garbage collection, the next garbage collection could bypass this area by marking only from the exit vector. Thus the LISP machine makes an attempt to reduce the number of areas involved in a garbage collection, but it is still necessary to include in each garbage collection all the areas that have been modified since the last garbage collection, thus this technique is not adequate for systems with a very large address space. ORSLA may wait until a significant number of modifications have been accumulated in an area before garbage collecting it and may reduce the size of individual garbage collections even among those areas that do have a significant number of modifications in order to eliminate thrashing in the garbage collector.

Areas on the LISP machine do not have a list of incoming references. This omission is a crucial difference between areas on ORSLA and the LISP machine. Another difference between ORSLA and the LISP machine is that no serious attempt is made on the LISP machine to guarantee the restrictions on the use of object references and thus the LISP machine cannot be considered to be a capability system. Although one of the several machine languages on the LISP machine does enforce the restrictions on the use of object references, much of the sophisticated system software, such as the garbage collector, cannot be written in this language. As with PL/I, it is assumed that the system programmer will naturally follow these restrictions. The approach on the LISP machine is better than that taken by PL/I, however, because most programmers will only need to write code that will be translated into the machine language that does enforce the restrictions on the use of object references. In addition, the software provided on the LISP machine, such as the garbage collector, assumes that the restrictions on the use of object references are being followed. Thus the user on the LISP machine will be penalized if he violates the inherent restrictions on the use of object references, while in PL/I he will not interfere with any of the system-provided software.

4.14.3 Baecker

It has also been proposed that areas be added to Algol 68 [Baecker72, Baecker75] so

that Algol 68 could deal with files better. An unusual aspect of this proposal is the suggestion that an area should consist of two parts: a table containing the pointer to and the size of each object in the area, and the storage containing the actual objects. At first glance this might appear to be a list of incoming links as on ORSLA, but on closer inspection it becomes clear that this is not Baecker's intention. Rather, the table is similar to the catalog of objects on a CUID system but done separately for each area. The concept of an offset within an area, which exists in PL/I and whose use is advocated by Baecker, would be implemented by an index into the table of objects for an area. The storage in the area could be compacted by changing the pointer in the table of objects but without changing the offsets of any of the objects in the area. If the immediately accessible objects in the area can be found and if the offsets stored within objects in the area to other objects in the area can be identified, then the area can be garbage collected. Baecker suggests garbage collecting the entire address space to find the accessible objects and uses the strong typing of Algol 68 to identify offsets in objects.

The areas on ORSLA differ from Baecker's areas in several respects. First, ORSLA always uses context independent object references that completely specify where in the system an object resides. Second, ORSLA keeps a list of only the objects in an area that are referenced from other areas. Since all the objects on this list are considered to be accessible on ORSLA, the storage for objects in the area could not be reclaimed if this list contained all of the objects in the area. Third, it is important that an inter-area link on ORSLA reside in the area the link is coming from, not the area containing the object it references (see page 84). Baecker's areas store the table of objects within the area containing those objects.

4.14.4 Lomet

Lomet [Lomet75] has proposed the construction of an "area machine" using a novel addressing scheme. Although what Lomet calls an "area" corresponds to an object on ORSLA, Lomet's areas are contained within segments which correspond roughly to areas on ORSLA. Lomet uses the tagged approach to capability systems and enforces the low level restrictions on the use of object references (which he calls "addresses"). His addressing scheme is a cross between a unique ID space and the linear paged address space proposed in

this thesis. He is able to place many objects on one page, but he still requires an entry in the object catalog (which he calls the area table) for each object. Lomet's addressing scheme pages as nicely as the linear paged address space on ORSLA, but requires more bits in the address than a unique ID space. Lomet uses tombstones to enable him to reuse sections of a page without having a reusable address space. Lomet does not have any mechanisms that are similar to the lists of inter-area links on ORSLA.

Chapter 5
Maintaining the Lists of Inter-area Links

The purpose of the lists of inter-area links on ORSLA is to provide each area with a complete list of the objects within the area that are referenced from other areas on the system. Given the list of incoming links to an area, the area can easily be garbage collected separately from the rest of the system by assuming that all the objects on this list are accessible. Since these lists are needed only by the garbage collector, it would be natural to assume that the garbage collector would be responsible for the maintenance of these lists. An unusual aspect of ORSLA, however, is that these lists are continuously maintained as a part of normal computation. Whenever an inter-area reference is created that is not covered by a cable, an inter-area link is created and threaded onto the appropriate lists. Thus the lists of inter-area links are always complete and accurate.

The correct maintenance of the lists of inter-area links can be achieved if every machine instruction is designed so that if the lists of inter-area links are accurate before execution of the instruction, they will be accurate after execution of the instruction as well. Such comprehensive design ensures the accuracy of the lists of inter-area links, but it does not ensure the practicality of the system. This chapter describes how this maintenance can be done without slowing down normal computation significantly.

5.1 Creation of Inter-Area References - What Must Be Computed

The first problem is to identify all of the ways in which an inter-area reference can be created. I approach this problem by considering the computer system at a very low level. There are only two basic suboperations of machine instructions that can cause an inter-area reference: load and store. Load gets an object reference from somewhere in the address space and places it in an internal register of the CPU. Store takes an object reference that is in a register and places it somewhere in the address space. The load and store suboperations of every machine instruction on ORSLA perform checks that determine when inter-area references are being created and construct the necessary inter-area links and cables.

If all inter-area references were done with inter-area links, the runtime overhead for checking all loads and stores and creating the necessary inter-area links would be very high. The main purpose of the cable is to reduce this runtime overhead. The existence of cables reduces the number of times inter-area links need to be constructed. The fact that cables are transitive reduces the overhead for checking load operations.

Exactly what checks need to be performed? It is only necessary to perform a check when an object reference is to be moved from one area to another. It must be remembered, however, that all of the CPU registers are considered to be part of the local computation area associated with the currently executing process. Thus object references may be moved from register to register without performing any checks. The only way an object reference can be moved from one area to another is through use of the load or store suboperations. In each of these suboperations there are as many as three areas (shown in Figure 16) that must be considered: area L, the local computation area, which contains the register being loaded or stored; area A, the area being accessed; and area B, the area pointed into by the object reference that is being moved between area A and area L. In addition there are two object references that are of concern: p, the contents of the register that points to the object in area A; and x, the object reference being moved between area A and area L.

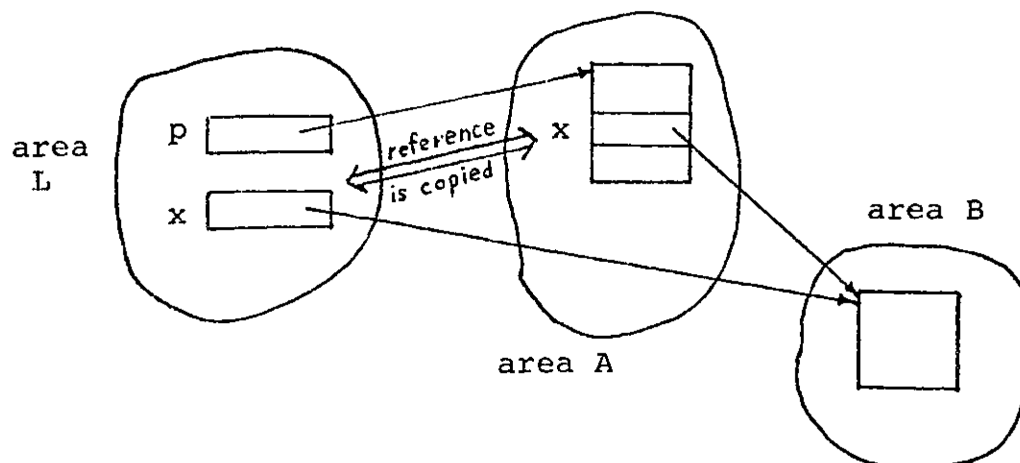


Fig. 16. Partial Notation for Load or Store

5.1.1 Load Operation

The load operation may also be concerned with three other object references if either p or x use inter-area links. In Figure 17, we see that p is the object reference in the register that points to the object in A. If it uses an inter-area link, then p' is the direct reference to the object in A. Similarly, x is the object reference actually stored in the object being accessed. If it uses an inter-area link, then x' is the direct reference to the object in area B. Finally, it may be necessary to construct an inter-area link in L. If so, x'' is the reference to this new inter-area link.

A flow-chart of the necessary checks for the load operation is given in Figure 18. The first question that must be asked is: does p use an inter-area link? In most cases, p will be a direct reference, so we will consider this avenue first. Since p is a direct reference from a CPU register to the object being accessed, we know that L is cabled to area A ($L \mapsto A$). The next question is: does x use an inter-area link? If not, and if x contains an address, then we also know that $A \mapsto B$, therefore, since the *cabled* relation is transitive, we know that $L \mapsto B$, so x may be moved to L without further action. This direct reference is covered by cables from L to B. If x does not contain an address it may be moved without further action, too.

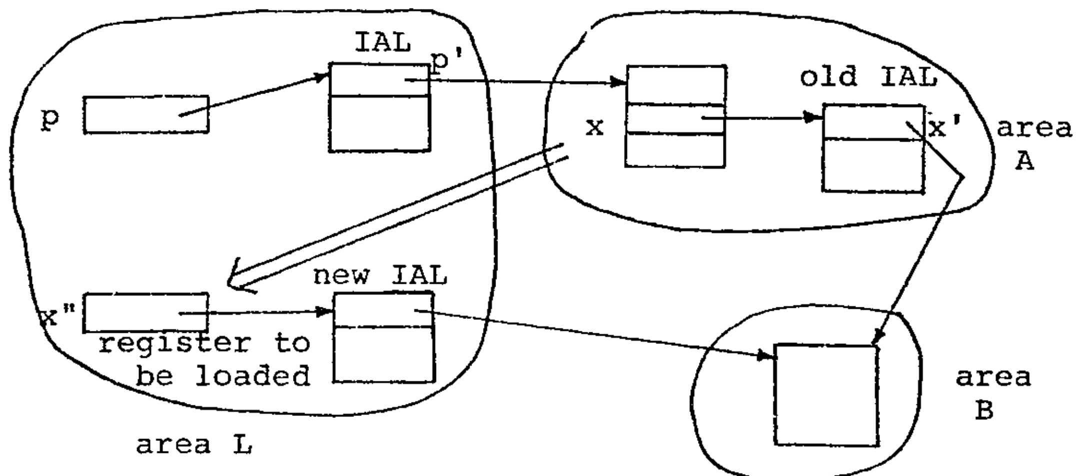


Fig. 17. Notation for Load Operation

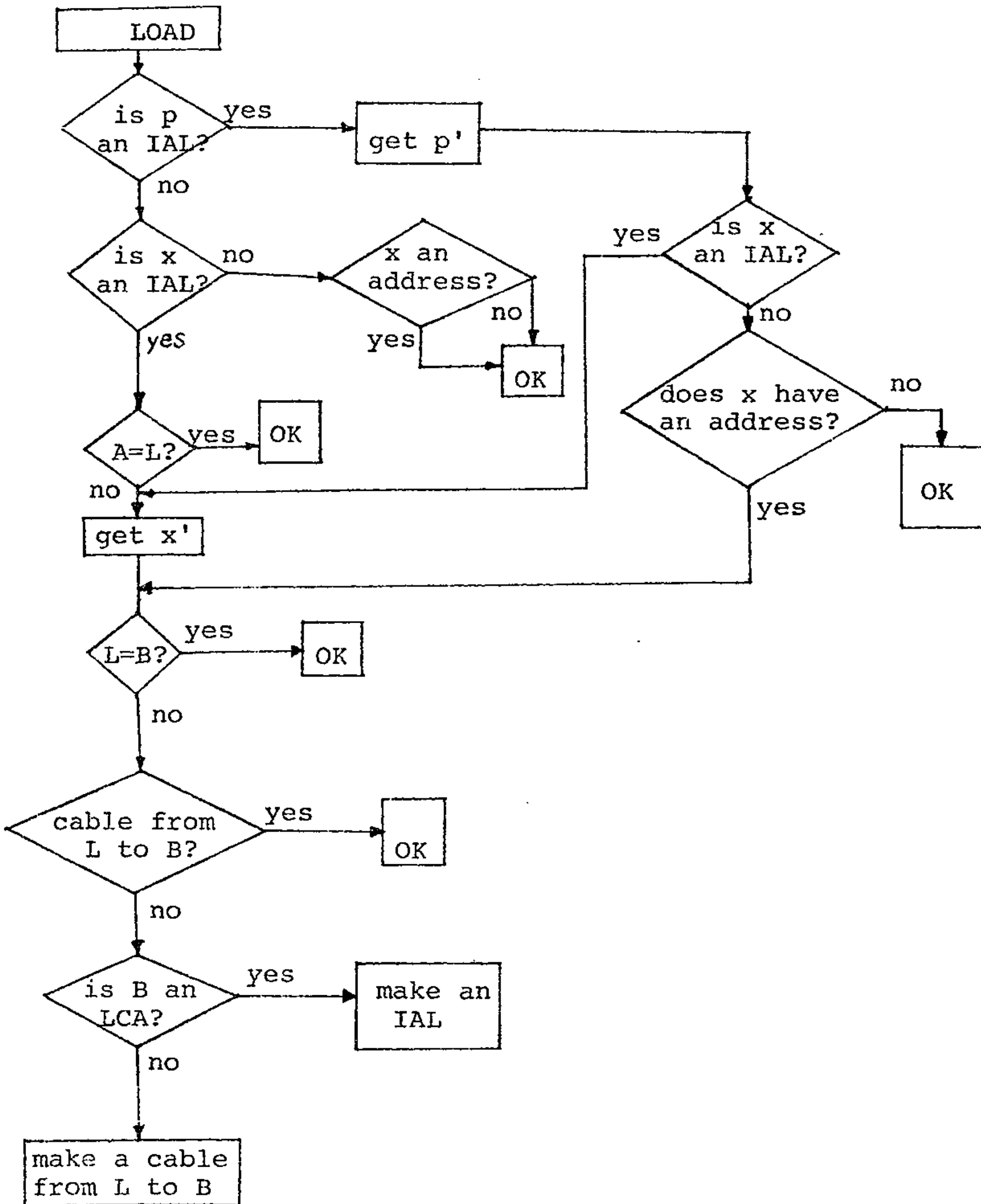


Fig. 18. Checks in the Load Operation

If x does use an inter-area link, however, more action is required. If area A = L, then presumably x is an inter-area link from L to another local computation area, but in any case, x, the reference to the inter-area link, may be copied within L without further action. If A $\neq$ L, then x, the reference to the inter-area link, may not be moved to another area. We must obtain x', the direct reference to the object in area B. Although L $\mapsto$ A, A $\not\mapsto$ B, so we do not know whether L $\mapsto$ B. If L = B, of course, then x will be an intra-area reference in L. If L $\not\mapsto$ B, then we must construct a cable from L to B unless B is another local computation area in which case we must construct an inter-area link from L to B and load the register with x'', the reference to this new link. If L $\mapsto$ B, of course, there is no need to construct either a cable or a link. Unfortunately it is very difficult to determine whether L $\mapsto$ B, since this cabling may go through several areas. Instead of doing this, then, we could determine whether there is a cable directly from L to B and if not, construct either a possibly redundant cable or an inter-area link.

We must now consider the case when p uses an inter-area link. In this case, we know that there is no cable from L to A. Presumably A is another local computation area. We also know that A $\neq$ L. Thus we cannot reduce the number of checks that must be performed. When no inter-area links are involved, the special properties of cables eliminate even the need for checking whether x contains an address. When p is an inter-area link, however, the full range of checks must be made whenever x contains an address. If x uses an inter-area link, then the main avenue of the load operation described above can access x' and perform the necessary checks. If x does not contain an address, however, no further checks need be made.

5.1.2 Store Operation

The notation used for the store operation is the same as the notation used for the load operation, thus Figure 16 describes some of the notation for the store operation. The complete notation for the store operation is shown in Figure 19. Although the complete notation for the load operation, shown in Figure 17, may seem to be different from that shown in Figure 19, on closer inspection it is apparent that the notation for the store operation is really the same as the notation for the load operation. In Figure 19, we see that

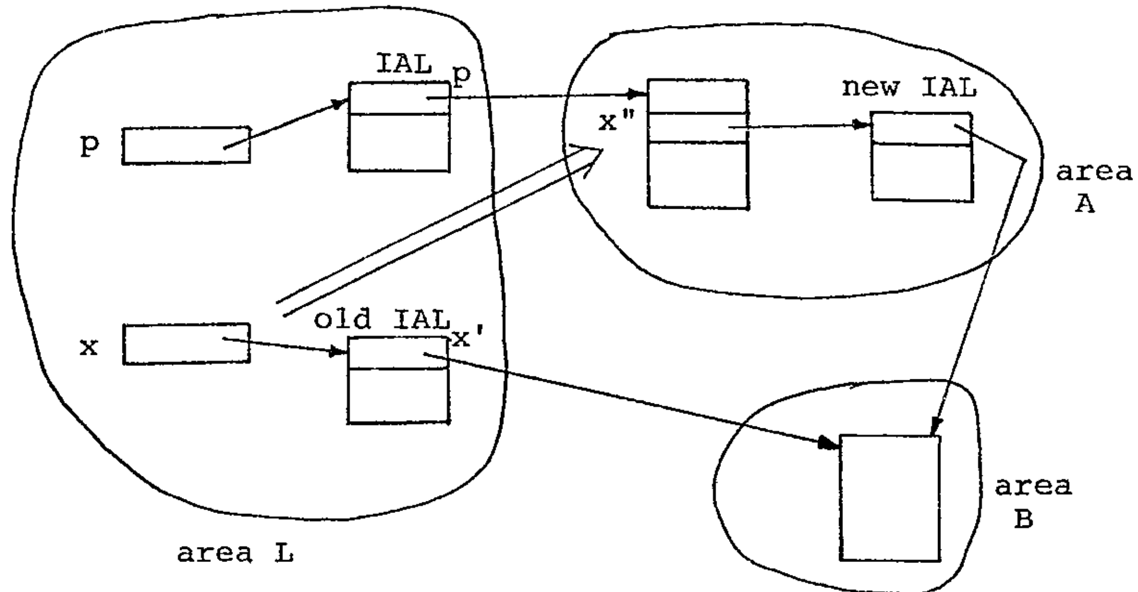


Fig. 19. Notation for Store Operation

---

p is the object reference in the register that points to the object in A. Similarly, x is the object reference in the register being stored. If it uses an inter-area link, then x' is the direct reference to the object in area B. Finally, it may be necessary to construct an inter-area link in area A. If so, x'' is the reference to this new inter-area link.

A flow-chart of the necessary checks for the store operation is shown in Figure 20. The store operation generally requires more checks than the load operation. If x does not contain an address, then no inter-area reference can be created and the store can be completed simply. If A = L, then x is being moved within L and no further action is necessary. Otherwise, however, we must perform a brute-force check. The fact that L may be cabled to area A and/or to area B has no bearing on whether area A is cabled to area B. Thus it makes little difference whether p or x use inter-area links. If p uses an inter-area link, the reference p' to the object must be retrieved. If x uses an inter-area link, the direct reference x' must be retrieved. If A ≠ B, then we check to see if there is a cable from A to B. If so, then x or x' can be stored into A without further operations. If there is

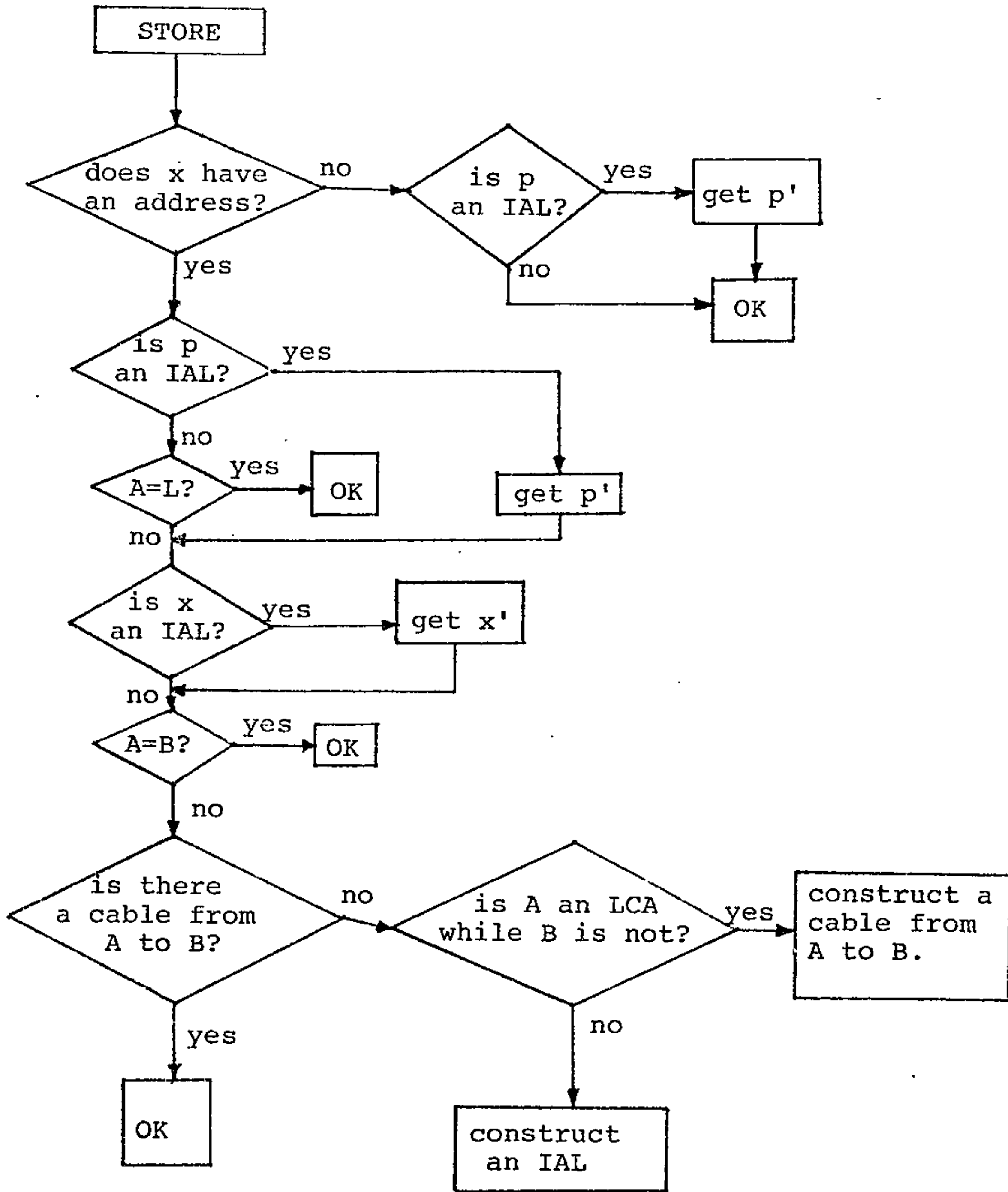


Fig. 20. Checks During the Store Operation

no cable from A to B, we check to see if A is a local computation area while B is not. If so, we construct a cable from A to B, otherwise we construct an inter-area link.

In order to construct an inter-area link or a cable from area A to area B, it is necessary to first lock both areas. Then the cable or link object can be allocated in area A and the link or cable threaded onto the list of outgoing links or cables in A and the list of incoming links or cables in B. Areas A and B can now be unlocked. The locking that is used must not interfere with higher level programs that are running in the process that may have locked either or both of areas A and B. If an area has already been locked by a process, then this process should be allowed to relock the area. If an area has been locked twice by a process, then it must be unlocked twice before the area will be truly unlocked.

5.2 Computing Checks for Load and Store Operations Efficiently

We have now seen what checks must be made before a simple load or store operation can be completed. The next question is: how can these checks be made inexpensive? The basic approach is to arrange for the operands of these checks to be available to the CPU without performing any additional memory references. Then the checks can be performed in parallel with the load or store operation itself. We must consider each step in turn and see whether it can be performed without additional memory references. It is not necessary for *all* of the checks to be done inexpensively, however. If, regardless of the outcome of a check, the following operation takes a long time, such as constructing an inter-area link, then one or two memory references to perform the check are acceptable. Even using an inter-area link requires an extra memory reference to access the direct reference within the link, so a little extra time used for checks in this case would be acceptable. The most frequent load and store operations, however, should not spend any extra time performing checks to maintain the lists of inter-area links. We have already seen, however, that the use of cables on ORSLA has eliminated the need for complicated checks on the load operation unless there are inter-area links. The use of cables does not eliminate any checks from the store operation, however, so most of the effort expended in computing checks efficiently should benefit the store operation.

5.2.1 Store Operation

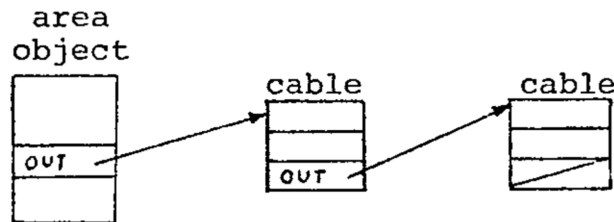
The first check that must be performed for the store operation is to determine whether x contains an address. As discussed in Chapters 3 and 7, the *data_type_info* field specifies whether x contains an address. Since x is already in a CPU register, the checking of this field is easy. If x does not contain an address, then no inter-area reference is being created by this store operation. It is still necessary, however, to check whether p uses an inter-area link. Since p is also in a CPU register, however, this can be done by merely checking the type code of p for type *inter-area_link*. If p does not use a link, the store can be completed without taking any extra time for these checks. If p does use an inter-area link, p' must be retrieved before we can compute the address being stored into. No extra time is required for checks, however.

Even if x does contain an address, it is necessary to check whether p uses an inter-area link. If p does not use an inter-area link, then the check (A = L?) must be performed. It would not be too hard to design the system so that a CPU register contained a reference to the area object for the local computation area (L), but getting a reference to A is not as easy. Area A is the area containing the object referenced by p. In order to perform the check (A = L), it must be possible to find a reference to A given p. ORSLA provides this ability by using a virtual memory catalog which, given a page number, can find the physical address of the page and also a reference to the area that the page is part of. This information is kept not only in the catalog, but also in the high speed map that maps virtual addresses to pages that are in high speed memory. A reference to area A can therefore be found at the same time that the physical address of the page is found. Since it is necessary to get the physical address of the page in A in order to perform the store, this check can also be performed without slowing down the CPU. The store operation is merely initiated, the reference to A obtained, and the check (A = L?) performed while x is being stored into A. If A = L, then the store operation can be completed without additional checks. In other words, as long as we store into the local computation area or store data that does not contain addresses, the CPU can perform store operations just as fast as if we were not automatically maintaining the lists of inter-area links. These two cases cover the majority of store operations.

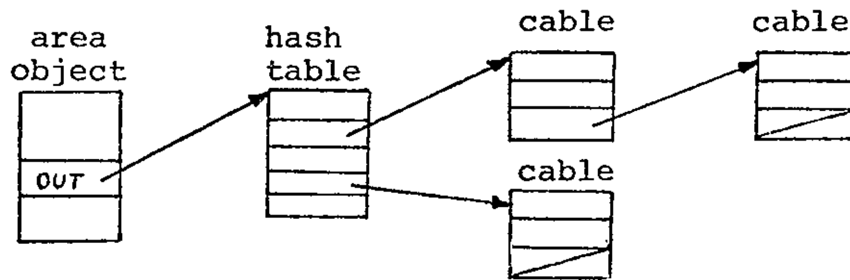
At this point we have only covered store operations to the local computation area. The checks from here on do not depend upon whether there are cables from L or not, so if either p or x use inter-area links, the direct reference (p' or x') must be retrieved. This involves overhead that would not be present if we did not have inter-area links. It is then necessary to initiate the store operation (using the address from p') and obtain the reference to area A. The next check in the store operation is (A = B?). B is the area containing the object referenced by x. A reference to B can be found by using x or x' (depending upon whether x uses an inter-area link) to access the virtual memory map. This operation would not be needed if we were not maintaining lists of inter-area links. On the other hand, it does not take much time; it takes about as long as a single memory reference. If x points to an object in A, then the store operation can be completed and the overhead is only about one extra memory reference. This case should cover most of the store operations of object references containing addresses that are stored outside of the local computation area.

If A $\neq$ B, then we have exhausted the simple cases. We know that we are storing an inter-area reference into an area other than the local computation area. Our only remaining hope is that there is a cable from A to B. This is determined by looking at the list of outgoing cables from A. Many areas do not have any outgoing cables. This could easily be discovered in one memory reference to the area object for A. If an area has a large number of outgoing cables, such as a local computation area, then searching the list of outgoing cables would be faster if the area maintained a hash table of outgoing cables in which each bucket pointed to a short list of outgoing cables (see Figure 21). In this way the average number of memory references needed to find a cable can be kept to less than 5.

If there is a cable from A to B, then the store operation can be completed without further effort. If there is no cable, however, then either an inter-area link or a cable must be constructed. If A is an LCA while B is not, then a cable must be constructed, otherwise an inter-area link must be constructed. Both of these operations require many memory references, so there is no need to optimize the test for whether an area is an LCA or not. The *area_information* field in the area object specifies whether the area is an LCA. The



An ordinary list is used for 0-3 cables. When the list is 3 cables long, it requires 7 memory references to find that there is no cable to a particular area, and an average of 4 memory references to find a cable that is present.



A hash table is used for more than 3 cables. The number of buckets in the hash table should be approximately equal to the number of outgoing cables. Thus on the average, there should be about 4 memory references to find a cable that is present and about 4 memory references to find that there is no cable. The estimates given here for average search time assume that all the cables are equally likely to be searched for and that the hash index is truly random.

Fig. 21. Hash Table of Outgoing Cables

---

combined test can be done with two memory references, one to the area object for A and one to the area object for B. Then both areas must be locked and the link or cable created and threaded onto the lists. This entire process should take about 20 memory references if fully supported by hardware or microcode, but it will probably all be done by software, in which case 40 or 50 memory references might be necessary. This is a major overhead that cannot be avoided. Hopefully, the construction of inter-area links and cables will be so rare that it will not be a significant source of overhead on ORSLA.

5.2.2 Load Operation

We have now seen the mechanisms used to compute efficiently the checks needed for

the store operation. Some of these mechanisms can also be used to reduce the time taken for some load operations. The most common kind of load operation occurs when neither p nor x use inter-area links (see Figure 17). In this case it is only necessary to check that neither p nor x use inter-area links, but these checks do not require any extra time since p is already in a CPU register and the load operation brings x into a CPU register. It is not necessary to check whether x contains an address because the load operation is performed simply in either case. Thus the most frequent load operations have no overhead for maintaining the lists of inter-area links.

If x uses an inter-area link, the direct reference within the link should be accessed unless the load is from L. The hardware required for the store operation can perform the check (A = L?) while x is being loaded so this check does not take any extra time. The last combination that avoids substantial overhead in the load operation for maintaining the lists of inter-area links is when p uses an inter-area link but x does not have an address. Once p' has been obtained and then x obtained, it can easily be determined that x has no address. The load operation can then be completed without having taken any extra time.

Finally we have reached a section of the load operation that must perform at least one extra memory access in order to perform checks that maintain the lists of inter-area links. The page map must be accessed to find an area specifier for B. This is only necessary if either p or x uses an inter-area link. Even then, if p uses an inter-area link but x does not contain an address, this section of the load operation will not be reached. The hardware developed for the store operation can easily perform the check (L = B?) which will allow the load operation to complete with very little overhead. Otherwise, it is necessary to check the cables from L to B. The hash table of outgoing cables developed by the store operation is very important at this point in the load operation because LCAs will probably have large numbers of outgoing cables. The hash table continues to average about 4 memory references per check regardless of the number of outgoing cables because the size of the hash table is merely increased to the size required to produce short lists in each bucket. If there is no cable from the LCA to the area, then either an inter-area link or cable must be constructed. The overhead in either case is large. Fortunately, this happens very rarely. Thus the major overhead for maintaining the lists of inter-area links

for the load operation consists of checking the cables from the LCA when p or x use an inter-area link and actually constructing links and cables.

An important check that reduces overhead in both the load and store operations is the check for a cable, so it is important for this check to be as fast as possible. Most operations on the list of outgoing cables must lock the area before proceeding whether the operation modifies the list of outgoing cables or merely inspects it. If, however, changes are made very carefully to the list of outgoing cables, then it is not necessary to lock the area when searching for an outgoing cable. It is not possible for such an unlocked search to be accurate if it is done while changes are being made to the list of outgoing cables, but it is possible to ensure that any errors that do occur consist of not finding a cable that is present. This is useful because if a cable is found, then the load or store operation is relatively short, while if no cable is found, the area must be locked to allow a link or cable to be constructed. Before actually constructing a link or cable, however, another search of the list of outgoing cables should be made. No error can occur during this second search because the area is locked. If no cable is found this time, then the link or cable can be constructed. By performing an unlocked search, two memory references are saved during searches that find a cable. If no cable is found, then the unlocked search is wasted at a cost of four memory references on the average. Since we expect most searches for cables to be successful, this trade-off is attractive.

An unlocked search can only be allowed if all changes to the list of outgoing cables are made very carefully so the list can be correctly read at all times. This can be achieved easily when adding a cable to the list by first filling in all the fields in the new cable before placing the reference to the new cable in the list of outgoing cables. Only a little more effort need be expended when changing the size of the hash table. In this case, the new table should be created and all the buckets in the new table should be initialized to "empty". Then the reference to the new table can be placed in the area object. Only then can we begin to rehash all the outgoing cables and add them one by one to the new hash table. Perhaps the most difficulty is experienced when deleting a cable whose *unused* bit is on. Remember that cables whose *unused* bit is off cannot be deleted at all. The critical condition occurs if an unlocked search for the cable finds it but does not turn off the

unused bit until after the cable has been deleted. This problem can be solved if we lock the area in order to turn off an *unused* bit in a cable. Thus if modifications to the list of outgoing links are made carefully, searches can be made quickly without locking the area.

Chapter 4 described the *unused* feature of explicitly created cables. When a cable is explicitly created by the user or the garbage collector, it is marked as *unused*. As long as the cable remains *unused*, it may be deleted explicitly and is not considered in the transitive closure of cables that defines the *cabled* relation. When a load or store operation searches for a cable and finds an *unused* cable, it turns off the *unused* mark and then creates a direct reference that depends upon that cable. The location of the *unused* mark in the cable object can affect the speed of the operation of finding a cable. If the *unused* mark is in the first word of the cable, which contains the object reference to the area that the cable goes to, then the *unused* mark can be checked at the same time this object reference is compared with the area we are searching for. The *unused* mark is stored within the object reference to the area that the cable goes to by using one state in the access control field of object references to areas to denote the *unused* mark. Thus the only cost for the *unused* cable feature is seven additional memory references the first time a direct reference is created that depends upon the cable. These seven memory references lock the area, find the cable again, and turn off the *unused* mark.

5.3 Special Hardware Needed on ORSLA

We have now seen what checks are necessary during load and store operations and generally how those checks can be performed without slowing down the computer system. It is easy to design a CPU to make checks using information that is already in the CPU without taking extra time. It is also easy to design into ORSLA a CPU register that identifies the current LCA. This section concentrates on how to identify the areas involved in load and store operations and how to perform some of the checks required by these operations.

5.3.1 Virtual Memory Mapping

It has been pointed out that a virtual memory system such as ORSLA must have a

high speed page map for mapping virtual addresses of pages in high speed memory into physical addresses. This map also holds an object reference for the area that each page is part of. Thus the area containing the location being accessed can be found as easily as the physical address of the location being accessed.

Cache memory on most virtual memory systems behaves like an associative memory that is a faster, smaller version of main memory, i.e. it takes a main memory address and returns the contents of the address. In order to perform a memory access using a virtual address, then, it is necessary to access the page map to convert the virtual address into a main memory address. Most virtual memory systems are designed so the page map is accessed in parallel with the cache since the information from the page map is only needed during the associative match in the cache. If such an organization is used on ORSLA, then the area containing the virtual address being accessed is available without additional work even if the associated main memory address is found in the cache.

There is another way to organize the cache that becomes possible on ORSLA since there is only a single address space in the entire system. It would be possible for the cache to map virtual addresses directly, thereby saving a little time since the associative match in the cache need not wait for information from the page map. In addition, it may be possible to reduce the time taken for purging information from the cache since the only way the information in the cache can become erroneous is if other processes write into the virtual addresses that are in the cache. If, on the other hand, the cache maps main memory addresses, then information must be purged from the cache whenever a page is moved out of high speed memory. Although it might appear that if the cache maps virtual addresses it is no longer necessary to access the page map as well as the cache, in fact it is still necessary to access the page map in parallel with the cache in order to find the area being accessed so that the necessary checks for maintaining the lists of inter-area links can be performed. The rest of section 6.3 considers how the hardware would be designed if this alternate organization for the cache were used.

The purpose of the cache is to provide good performance at a reasonable cost, but the system can function without it. The page map, however, is essential to the operation of

virtual memory. Usually it will be economical to use two different kinds of high speed memory on ORSLA: main memory with an access time of 500 nanoseconds or more, and very high speed memory with an access time of 100 nanoseconds or less for the cache and the page map. If only one speed of high speed memory is available, ORSLA would be built without a cache but with a high speed page map.

5.3.2 Page Map

Main memory is broken into pages. Each page of main memory is dynamically assigned to a page of virtual address space. The CPU always deals with virtual addresses. Every memory access must convert a virtual address into a storage address. This is done by a special unit called the page map (See Figure 22). On many systems the page map contains the locations of only 16 pages of virtual address space [Schroeder71]. In order to access the rest of the pages in main memory, it is necessary for these systems to have an additional table in main memory that specifies the locations of all the virtual pages that are in main memory. ORSLA, however, uses a page map that is large enough to map all of the virtual pages that are in main memory: probably between 100 and 5000 pages.

The page map must take a virtual address and produce a main memory address. The virtual address is broken into two parts: a virtual page number and an offset within a

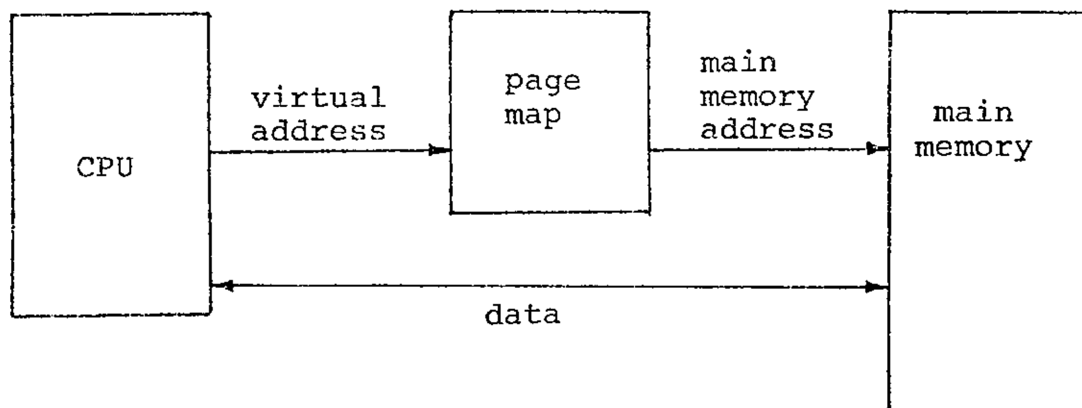


Fig. 22. Page Map Overview

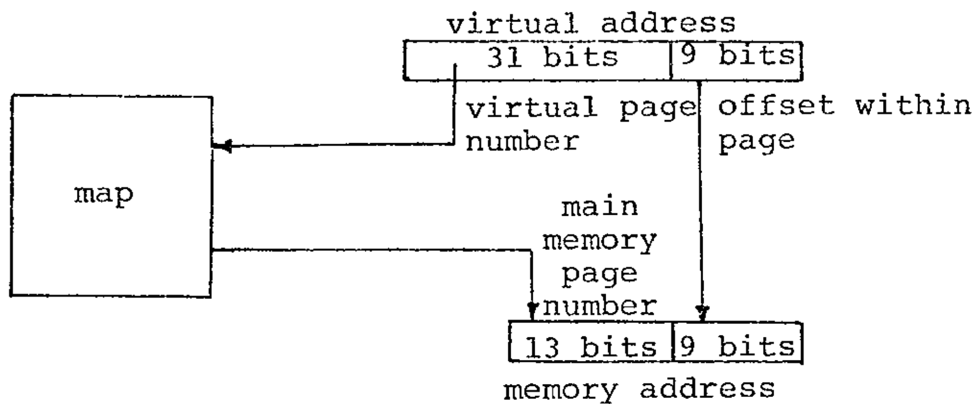


Fig. 23. Mapping a Virtual Address

page (See Figure 23). The memory address is formed by finding the main memory page number that corresponds to the virtual page number and appending to it the offset within the page from the virtual address. The essential part of the page map is the memory that converts a virtual page number into a main memory page number. Ideally, the page map memory is an associative memory in which the contents are accessed by virtual page number. Since the mapping memory must be accessed before memory can be accessed, the effective access time of main memory is increased by the access time of the mapping memory. If the mapping memory uses very high speed memory, then it will have an access time of 100 nanoseconds while the main memory has an access time of 500 nanoseconds, resulting in an effective access time of 600 nanoseconds to main memory.

The mapping memory must contain other information as well, however. Usage information must be kept on each page in main memory so that ORSLA can determine which page to swap out of main memory. Usually this is just one bit that system software turns off every now and then and which is automatically turned on whenever the page is accessed. ORSLA has a special need, however. It is necessary for this page map to contain an object reference for the area each page is part of. It is not necessary for a full-size object reference to the area object to be used, because most of the bits will be constant. The type code always specifies type *system_area*; the *size* field and miscellaneous information are always the same. Since the area object is the first object placed in an area,

the offset from the beginning of the page is a constant. Thus the only variable information in a reference to an area is the page number in which the area object resides, which only requires 31 bits in a 40 bit address space with 512 word pages.

In addition, two other bits of information are needed in the page map. First, the garbage collector needs to know whether an area is being garbage collected, so there is a *GC* bit. The second bit, the *deleted* bit, is needed to specify whether the area has been deleted. The need for this bit will not be apparent until Chapter 6, however. Although some of these bits duplicate bits in the *area_information* field of an area, there is no *deleted* bit there, and the *LCA* bit, which is in the *area_information* field, is not kept in the page map. Information is kept in the page map only if it can significantly shorten operations that are used very frequently. Since placing the *LCA* bit in the page map would not significantly shorten the long operations in which it is used, such as constructing a link or a cable, the *LCA* bit is not in the page map.

Thus the extra information needed by ORSLA in the page map does not quite double the size of a page map entry. Ideally, all of this information would be contained in each element of an associative memory that can be accessed with the virtual page number (See Figure 24). Ideally, there would be as many elements in the associative memory as there are pages of main memory.

Associative memories of this size become slow, however. The effect of an associative memory can be approximated by a set associative memory. The IBM 370/165 uses a set associative memory with four elements in each set and LRU replacement within a set

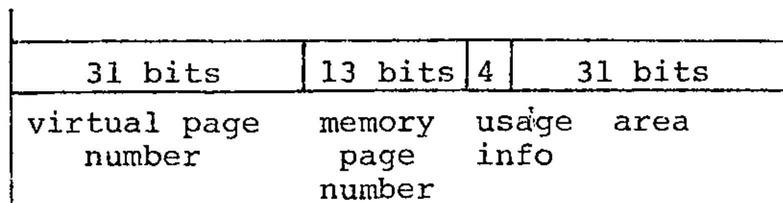


Fig. 24. Page Map Entry

[Madnick74]. The IBM 370/165 uses this memory for cache, but this type of memory could easily be used for a mapping memory as well. A set associative memory with four elements per set does a good job of approximating a full associative memory. If ORSLA could use a true associative memory, there would be no need for it to have a replacement algorithm since it would always contain the complete data base. If a set associative memory is used, however, it is possible for more than four pages to be in the same set, in which case their addresses cannot all be placed in the fast memory. It is necessary to have the full page table in main memory and for a mechanism to exist that will load the mapping memory when a page in main memory is accessed whose mapping entry is not in the mapping memory.

How well does a set associative memory with four elements per set approximate a true associative memory? Let us assume that we have a 1000 element memory with 250 sets and that there is a random set of x pages all of which we would like to have in the mapping memory at the same time because they all form an important part of the working set. If we assume that the probability of each page of virtual address space being a member of a set is uniform, then if x is less than 225 pages, then the probability that the entire set of x pages will all be in the mapping memory is greater than 0.5[1]. Thus a set-associative memory approximates a true associative memory rather well.

Even if it is practical to have a large mapping memory, it may be economical to have a smaller mapping memory. Multics proved the feasibility of using a small mapping memory. Fabry [Fabry74] has suggested that a small mapping memory be used for CUID systems even though these systems have a single address space as does ORSLA and so could avoid making changes in the mapping memory except when swapping objects if the mapping memory were large enough. ORSLA has more to gain from a large mapping memory than does Multics. Furthermore, ORSLA cannot operate with as small a mapping memory as can Multics because ORSLA must make additional accesses to the mapping memory in order to perform checks to maintain the lists of inter-area links. Thus ORSLA

---

1. This number was arrived at theoretically but was verified with a Monte Carlo method.

must operate with a larger mapping memory than Multics. Since ORSLA maps pages while CUID systems map objects, however, CUID systems will need an even larger mapping memory than does ORSLA.

Once the page map specifies what area each page is part of, the operation of finding the area that an object is in is very inexpensive. We are now ready to consider how we will make use of these area specifiers. In particular, we are interested to see if there will be any other structural changes in the page map in order to make effective use of the area specifiers. The area specifiers were introduced to speed up the checking needed for the store operation, so we will consider how well the needs of the store operation have been met with the hardware that has been proposed so far and see if any further hardware is needed to meet the needs of the store operation.

Before we consider what checks must be made and how to make them, let us review the notation used in the previous discussion of these checks (see Figure 16). The object reference that specifies the object being stored into is p. The actual address is calculated from p. The location pointed at by p is in area A. The object reference being stored is x. The area containing the object referenced by x is area B. L is the local computation area.

The CPU performs the first couple of checks in the store operation: checking p for an inter-area link and checking x for an address (see the flow chart in Figure 20). If x does not contain an address, no further checking is necessary, so there is a control line from the CPU to the page map that specifies whether checking should be done. If checking is necessary, then the first check that must be performed is to see whether A is equal to L. The page map should contain a register (the LCA register) that contains the area specifier for L (see Figure 25). The LCA register is loaded whenever the processor switches processes. When a store operation is signalled, the page map finds the main memory location of the page being stored into and the area specifier for A. The area specifier for A is compared with the contents of the LCA register. If they are equal, no further checks are necessary. If not, the rest of the checks are performed, but the store may proceed. The result of the comparison of L and A is reported to the CPU in case x is an inter-area link, so the CPU can abort the store and take further action. Otherwise,

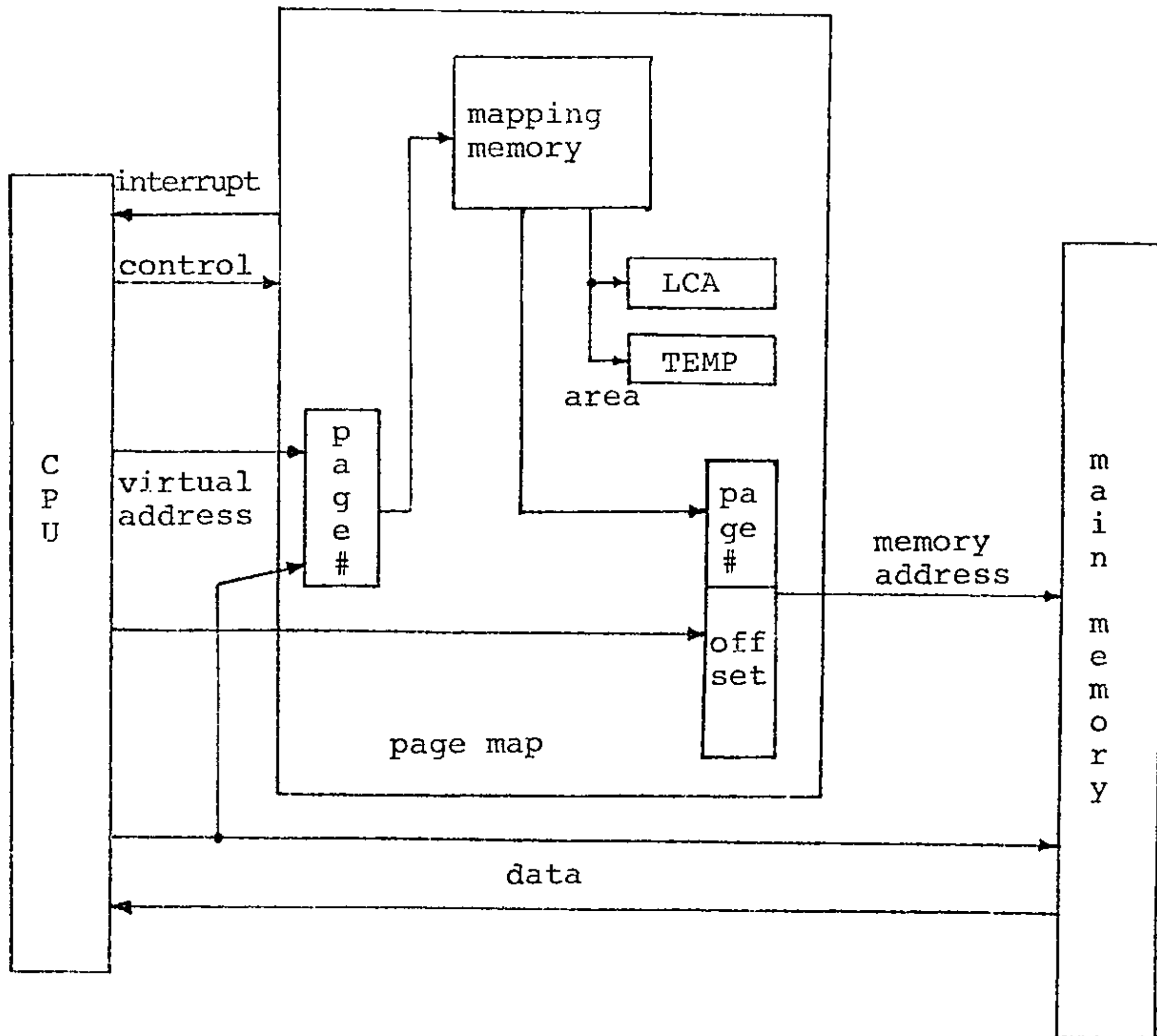


Fig. 25. Page Map in Detail

---

checking continues in the page map itself. The next check determines whether A is equal to B. The specifier for A that was just obtained must be saved in a temporary register (the TEMP register). The page map now gets the address from x and makes an access to the mapping memory again. If the page map does not contain an entry for this page, a page fault must be generated and the store operation must be aborted. Otherwise, the specifier for B is returned and can be compared with the contents of the TEMP register. If A = B,

then no further checks are needed. Furthermore, the access to the page map to find the area specifier for B was able to proceed in parallel with the store operation to main memory. Thus even though the check for A = B is extra computation, it may be accomplished without slowing down normal computation. By placing the checks for A = L and A = B in the page map, we have eliminated the overhead for maintaining the lists of inter-area links when storing anything in the LCA or when storing intra-area references outside of the LCA.

If A ≠ B, however, then much more complicated checks need to be made, so the store should be aborted. The other checks that need to be made by the CPU depend upon A and B, so the specifier for B is placed in the TEMP register and the specifier for A is sent to the CPU. When area specifiers are sent to the CPU from the page map, the rest of the bits are supplied to convert them to object references for area objects. The remaining checks are complicated enough and rare enough that they can be performed by microcode or by software; the overhead for these checks can be estimated in a straightforward manner.

5.3.3 Cache

Cache memory is made of very high speed memory, having an access time less than 100 nanoseconds. An unusual feature of ORSLA is that the cache can map virtual addresses instead of main memory addresses. Such an organization is an intriguing possibility on a system with a single address space, but there are many reasons why main memory addresses are used in cache memories. It is beyond the scope of this thesis to consider all these issues, but it is my responsibility to show that no special feature of ORSLA makes it more difficult to use a cache. After reading this section, it will be obvious how to design a cache for ORSLA that uses main memory addresses. Since the design of the cache is less obvious if it uses virtual addresses, however, I discuss the design of this type of cache in this section.

Conceptually the cache memory on ORSLA is an associative memory which, given a virtual address, can return the contents of that address. Each entry in the cache actually contains more than one word, however. Depending upon how the cache and main memory

40-50 bits	59-82 bits/word	4
virtual address	1-4 words	misc bits

Cache memory has 1-4 words per entry depending upon system parameters.
Fig. 26. Cache Memory Entry

have been designed, each entry in the cache contains either one, two, or four words. Only one of these sizes is used in any particular system, of course. Each entry in the cache also contains a few bits that are necessary for the maintenance of the cache and for the realization of LRU replacement. The format of a cache entry is shown in Figure 26. Conceptually the cache is a pure associative memory, but since each entry is so small, the cache would have to be a large associative memory. Large associative memories, however, are both expensive and slow, so a pure associative memory is approximated by a set-associative memory. This is a standard technique in the design of cache memories.

Figure 27 shows a set-associative cache memory with 4 entries per set and four words per entry. The virtual address is broken into three pieces: the offset (*o*) within a cache entry, the set specifier (*s*), and the rest of the virtual address (*v*) which must be associatively matched against the address of an entry. The cache memory consists of two memories that are accessed in parallel: the address memory and the contents memory. A word in the address memory contains the four addresses for the four entries in one set; it is accessed with the set specifier part of the virtual address. A word in the contents memory holds the contents of four virtual addresses, each address being at offset *o* within each entry in the set. The contents memory is accessed by the set specifier and offset portions of the virtual address. Thus both the address and contents memories can be accessed given only a virtual address. An associative match is performed on the information retrieved from the address memory with the unused portion (*v*) of the virtual address. If the virtual address matches one of the entiries in the set, then the word in that entry from the contents memory is gated into the READ buffer.

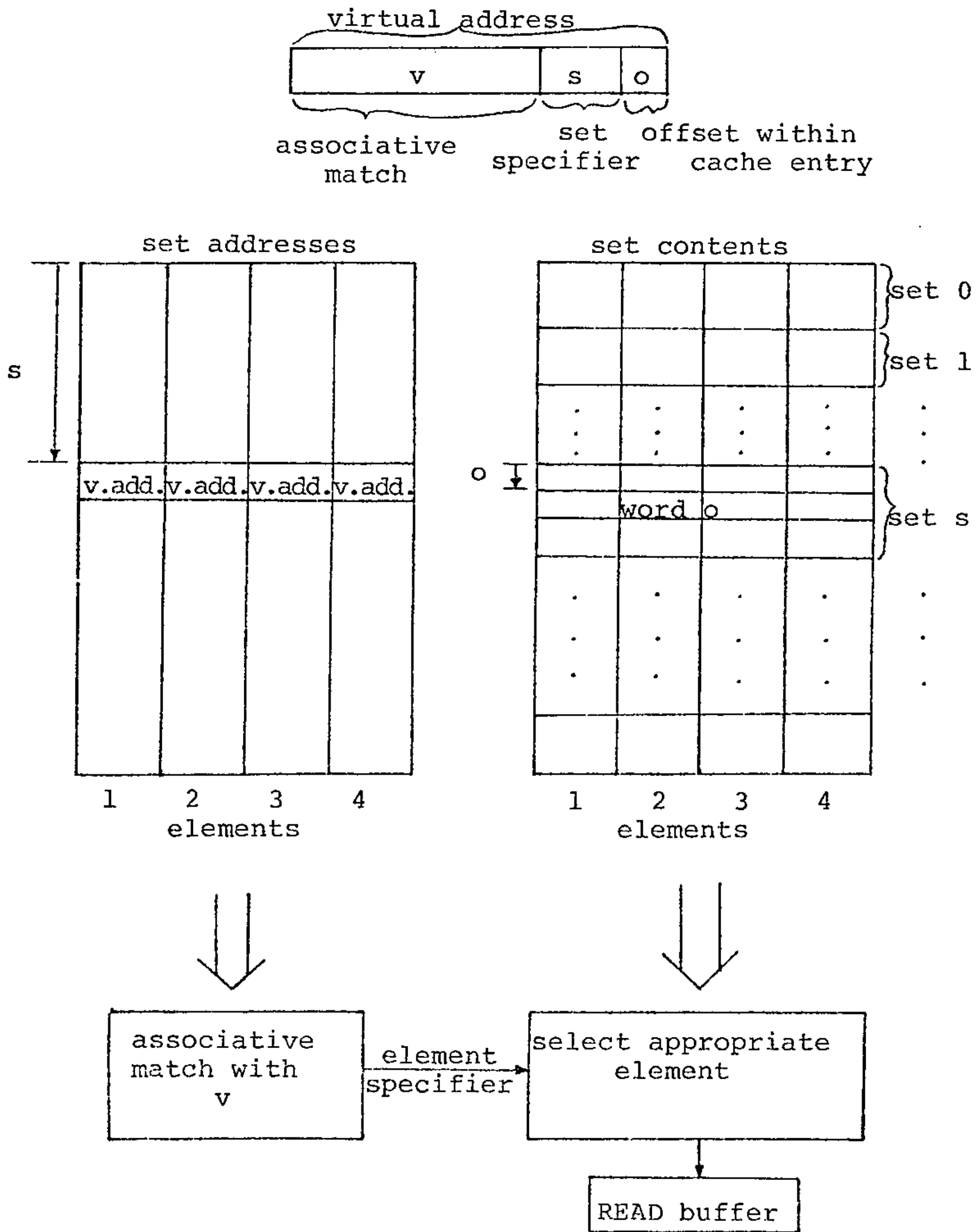


Fig. 27. Cache does Read before Write

The page map is accessed in parallel with the cache to find what area the virtual address is in and to find the associated main memory address. Since the page map uses a set-associative memory of the same type as the cache, the main memory address is completely prepared[1] by the time it is discovered that the virtual address is not in the cache. Thus when a cache is used, no extra time is taken by the system for virtual memory mapping.

A feature of ORSLA that might affect the structure of the cache is storage monitoring. Checking for a monitor in a location being stored into requires that the previous contents of the location be read before the write takes place. Most cache memory systems do not use the cache for store operations, however; they use "store-through" directly to main memory. A cache that is organized in this way is not involved in the implementation of storage monitoring; instead, main memory must always perform store operations with a read-modify-write cycle during which storage monitoring is implemented. If, however, a cache memory allows store operations to be performed directly to the cache without performing a "store-through" to main memory, what mechanism would be needed in the cache to implement storage monitoring? The answer is that surprisingly little is needed to implement storage monitoring. A store operation to the cache must first access the address memory to find whether the virtual address being stored into is in the cache. The contents memory is normally accessed at the same time as the address memory, and if this is done even for store operations, then the contents of the virtual address are read by the cache without taking any extra time. After the associative match has found which entry in the set contains the virtual address, then the appropriate part of the cache can be written into. Practically the only modification that is needed to the cache to implement storage monitoring is to provide a double set of buffers: the READ buffer to hold the previous contents of the accessed word and the WRITE buffer to hold the information being written into the cache. Retaining the previous contents of a word also allows a store

---

1. If the page map does not contain an entry for this virtual page, the complete map of the pages that are in high speed memory must be accessed by the CPU. If the page is not in high speed memory a page fault is generated.

operation to be aborted after it has begun.

We are now ready to consider how the cache memory works with the page map to perform a store operation, assuming the cache does not use "store-through" to main memory (see Figure 28). The CPU begins a store operation by sending the address to both

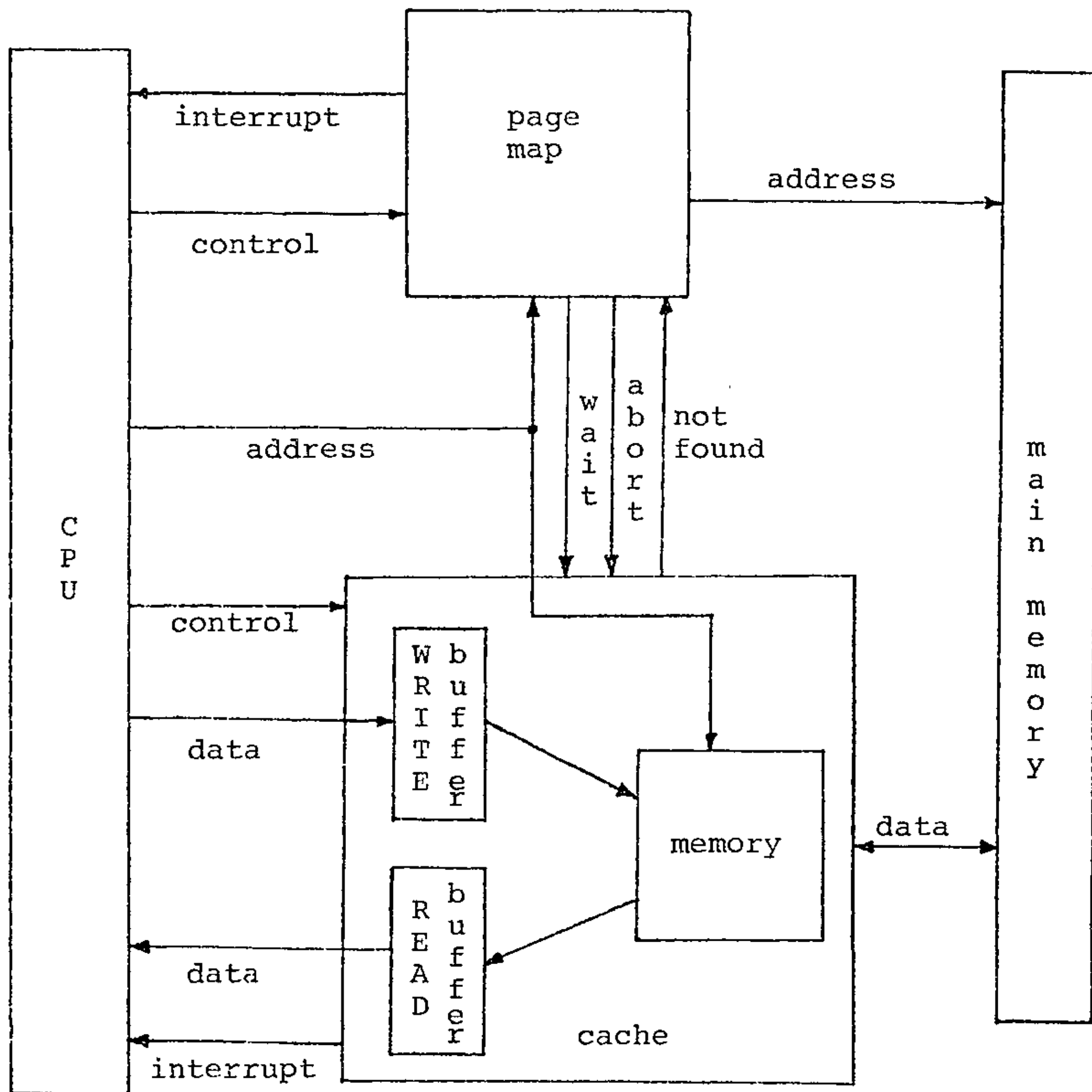


Fig. 28. Cache Memory in Detail

the cache and the page map. The cache and the mapping memory are accessed in parallel. If the address is not in the cache, the cache sends a signal to the page map to initiate a main memory cycle. By this time the page map has been accessed so that when the signal for a main memory cycle reaches the page map, the main memory address has been prepared or a page fault has been signalled to the CPU.

If the address is in the cache, then the only purpose for accessing the mapping memory is to perform checks to maintain the lists of inter-area links. If $A \neq L$, then the page map sends a signal to the cache to wait before signalling completion of the store operation. This signal reaches the cache at the time that the cache has just recognized that the address is contained in the cache. While the page map is performing a second cycle to find the area specifier for B, the cache is actually storing the data into the appropriate location in the cache. Much of the second cycle in the page map is complete by the time the store has completed. It will probably take a little longer for the second page map cycle than for the store operation, so there will be a little overhead for such a check, but it will probably not exceed 25% of the total time taken for a store operation. As before, the page map makes its checks automatically. As a result of these checks, it may be necessary to abort the store operation so more complicated checks can be made by the CPU. However, the vast majority of store operations that do not involve inter-area links or cause page faults will be handled directly by the checks made in the page map.

Thus we see that by carefully designing the hardware on ORSLA, it is possible to reduce the overhead for maintaining the lists of inter-area links almost to zero. Furthermore, the complexity of the additions to the hardware needed for ORSLA is less than that needed to implement virtual memory. In fact, ORSLA is able to make more effective use of virtual memory than do SAV or LAV systems.

5.4 Comparison of ORSLA with Other Systems

We have now seen how the load and store operations are performed on ORSLA. Individual load and store operations have an overhead that ranges from zero to a factor of 50. This overhead is only meaningful in comparison with other systems, however. In this section, I compare the speed of the load and store instructions on ORSLA with the

speed of the equivalent operations on systems whose addressing hardware is used by object references in programming languages. Thus I do not consider CUID systems in this section but do consider the B6700, SAV systems and LAV systems. Programming style is affected by the structure of the system, so there are different programming styles on the B6700, SAV systems and LAV systems. During each comparison, I try to show how the programming style on each system would naturally be transformed into the corresponding programming style on ORSLA. We will see that the loads and stores on ORSLA that correspond to loads and stores using the address space on SAV and LAV systems have very little overhead, so we expect the programming styles used on SAV and LAV systems to run just as fast on ORSLA as they do on their own systems. Many of the loads and stores on ORSLA that are not supported by hardware on SAV or LAV systems involve little overhead, however, so we expect a new programming style to be developed on ORSLA that will take advantage of these inexpensive operations to increase the performance of the system. Thus the apparent overhead that exists on ORSLA for maintaining the lists of inter-area links does not slow down ORSLA. Instead, it allows the use of a single address space for the entire system which is an inherently more efficient address space than those found on SAV or LAV systems.

The comparison of ORSLA to the B6700 is very brief because the B6700 discourages the copying of object references, while they can be freely copied on ORSLA. The overhead on ORSLA for maintaining lists of inter-area links occurs only when object references containing addresses are being copied. If ORSLA were used in the way the B6700 is used, very little overhead due to the maintenance of inter-area links would be invoked, so ORSLA compares favorably with the B6700.

SAV and LAV systems can manipulate addresses very easily, so it would appear that the overhead for maintaining inter-area links slows ORSLA down relative to these systems. We will discover, however, that the overhead for maintaining inter-area links occurs during operations that correspond to operations on these other systems that are not adequately supported by the hardware of these other systems, so these operations are costly on these other systems as well. Both SAV and LAV systems have several types of storage that are used in different ways. In order to compare the loads and stores on these systems,

we must identify the different kinds of storage so that we can determine which loads and stores on the three systems accomplish the same tasks. These storage types are: temporary storage (T), external temporary storage (ET), machine code storage (C), and permanent storage (P).

The first kind of storage that is treated specially by all three systems is the temporary storage for a process. This storage contains the per-process information and intermediate results of the process. On ORSLA, it is the local computation area (LCA). Loads and stores to the LCA never have any overhead for maintaining the lists of inter-area links. Furthermore, object references to anywhere in the system can easily be kept in the LCA. On LAV systems, such as Multics, temporary storage is contained in the segments in the process directory. Pointers are used extensively to organize information in temporary storage and are also used to point to most other storage on the system as well. On SAV systems, however, all of the writeable portions of the address space of a job are in temporary storage. Pointers can easily be used within this temporary storage, but may not point outside of the address space of the job. Occasionally, jobs on SAV systems use disk for temporary storage, but this is done only when there is not enough room in the address space. Since computations that are well suited for SAV systems do not use all of the address space, this source of inefficiency will be ignored here. Thus all the temporary storage for a job on an SAV system is contained in the address space of the job.

The rest of the address space for a job on an SAV system contains the machine language programs that are needed by the job. This special handling of machine code on SAV systems causes us to consider the storage that holds the machine code to be another kind of storage. It is possible for pointers within the temporary storage of a job to point into the machine code storage for the job. Pointers within machine code storage may point at other machine code programs within the job. On ORSLA and LAV systems, however, storage containing machine code is very similar to other permanent storage and is not kept in the LCA or the process directory. On ORSLA, programs can contain direct object references for constants and procedures within the same area, but must use inter-area links for procedures in other areas. On Multics, a machine language program may use intra-segment pointers within the program, but must use dynamic links to procedures in

other segments. These dynamic links are placed in the process directory automatically and the first time they are referenced, they cause a fault which is recognized by the system which then finds the proper pointer and stores it in the dynamic link. This is called "snapping" the link. The link may then be used without overhead for the rest of the process.

The overhead associated with dynamic links on Multics has some similarity to the overhead for maintaining and using inter-area links on ORSLA. The first time one of the programs in an area is called, a cable must be constructed to the area from the LCA. This is similar to the operation of making a segment "known" to a process on Multics. A segment number is assigned to the segment and a linkage section is created for the segment and is then initialized. The linkage section contains all the "internal static" storage for the programs in the segment and all the dynamic links from this segment to other segments, although these dynamic links are initially unsnapped. Making a segment known to a process on Multics clearly takes longer than constructing a cable from the LCA on ORSLA. Once a cable has been constructed from the LCA to an area on ORSLA, then whenever an inter-area link to this area is used, the LCA must be checked to make sure there is a cable to the area. On Multics, on the other hand, the first time a dynamic link is used, it must be snapped, which is a very long operation. Once a link has been snapped, however, it can be used without further overhead. The total overhead on the two systems is probably comparable, however, since the extra overhead on Multics for making a segment known to a process and snapping each dynamic link may balance the extra overhead on ORSLA for checking for the existence of a cable from the LCA whenever an inter-area link is used.

There is no overhead on SAV systems that corresponds to using an inter-area link or a dynamic link as long as the programs that are called reside within the address space of the job on an SAV system. Since the address space on an SAV system is about as large as a segment on Multics or an area on ORSLA, it is possible for a programming style developed on an SAV system to be transfered to Multics (or ORSLA) by placing all the procedures found in the address space of the SAV system into a single segment (or area) so that dynamic links (or inter-area links) need not be used between these procedures. SAV

systems sometimes provide dynamic linking as well as the static linking within the address space that is usually used. Dynamic linking on SAV systems is used when a program is being called that does not reside within the address space of a job. The required program is read into the address space and all external symbols (except for other dynamic links) in the program are linked to objects within the address space. This is sometimes called "loading" and has much more overhead than dynamic linking on Multics or the use of inter-area links on ORSLA. Once a program has been loaded, however, it may be used without further overhead. Thus it is not clear that Multics or ORSLA requires more overhead than SAV systems for procedure calls, but there is some overhead for procedure calls in all of these systems. The overhead on SAV and LAV systems is caused by dynamic linking which is needed because of limitations of the address spaces on these systems, while on ORSLA, it is caused by the need to maintain the lists of inter-area links.

The third kind of storage on these systems is permanent storage for information other than machine code, e.g. source code, other text files, data files, and complex data bases such as those used for airline reservation systems. This information is almost always kept on disk. On SAV systems, there are two ways to use such information. A permanent file can be copied into the address space of a job so it can be used heavily, or it can be used where it is with explicit I/O commands to obtain the parts of the file that are needed when they are needed. Thus a program that uses a permanent file is written differently depending upon what the total pattern of access to the file is from the entire computation. Furthermore, a permanent file that will be modified by several parallel computations, such as an airline reservation system data base, cannot be copied into the address space even if it is being used heavily, because this creates a copy of the file which may not be modified without modifying the original file and all other copies of the file that have been created by other processes using the file. Instead of attemping to keep all the copies of a file consistent, the copy on disk is used with explicit I/O commands. On LAV systems and ORSLA, however, the permanent storage used by a computation is within the address space and so information within it is used in the same way as information in temporary storage: by having object references to objects in permanent storage and using these object references to obtain information from the object. Thus SAV systems do not support the use of object references to objects in permanent storage. Even if software on an SAV

system did allow some form of object reference to objects in permanent storage, the most fundamental problem of SAV systems is that "addresses" to objects in permanent storage must be interpreted by software. LAV systems and ORSLA provide hardware supported address spaces that allow efficient use of object references to objects in permanent storage. There are limitations on the usefulness of the address space on an LAV system, however. LAV systems do not allow full object references to be stored within objects in permanent storage. Since permanent objects will be used by many different processes, each with its own address space on an LAV system, full addresses cannot be stored meaningfully in permanent objects. Many LAV systems support an intra-segment pointer, however, that allows a different format of object reference to be used within permanent storage than is used in temporary storage. The new format of object reference can only point to objects within the segment in which it resides, however. Thus LAV systems as well as SAV systems use a different representation for objects in permanent storage than is used for objects in temporary storage, forcing programs to choose whether to operate on objects in permanent storage or on objects in temporary storage. On ORSLA, the same representation for objects may be used in temporary storage and permanent storage, allowing programs originally written to operate on temporary objects to be used on permanent objects as well.

The fourth kind of storage is external temporary storage. This is the temporary storage for another process. External temporary storage is accessed to perform inter-process communication. On many SAV systems, the address space for another job is not treated as a file in the file system and so special inter-process communication primitives are provided by the system. Whether the address space for another job is handled via I/O operations or special inter-process communication primitives, external temporary storage on SAV systems is manipulated with a software supported address space which has significant overhead. On Multics, users do not have access to each other's process directories, so external temporary storage cannot be accessed through the address space. Special inter-process communication primitives are used on Multics as well to allow the flow of information between processes. Inter-process communication software is a form of software supported address space, and so slows down both SAV and LAV systems. In addition, it is not possible to communicate object references between processes on these systems because the processes use different address spaces. On ORSLA, one process may access the temporary

storage of another process as long as the first process has a reference to an object in the temporary storage of the second process. Object references can be communicated between processes on ORSLA since they share the same address space. ORSLA does have significant overhead for these references, however, since they must use inter-area links.

Another way of characterizing a load or store is by the kind of information that is moved. Thus a load or store can be characterized by the type of storage accessed and the kind of information moved. The easiest kind of information to manipulate is atomic information (A) that does not contain any pointers. Any difficulty with dealing with atomic information is attributable to the kind of storage being accessed, not the nature of the data being moved.

Pointers are harder to deal with. The first kind of pointer is the internal pointer (IP). An internal pointer in the temporary storage for a process points within temporary storage for that process. This is the most common kind of pointer and it can be handled relatively easily by ORSLA, SAV systems, and LAV systems. Another kind of pointer is the external pointer (EP), which points from one area to another. There are three different kinds of external pointer, depending upon the type of storage being pointed at: temporary (EPT), machine code (EPC), or permanent (EPP). Of these three types of pointer, the one that is most frequently given the most system support is the EPC pointer when it is stored in machine code (pointer to a subroutine). The other types of pointers are seldom given much system support except on ORSLA.

Table 2 compares the difficulty of load and store operations on ORSLA to those on SAV systems and LAV systems. A line in the table stands for a specific kind of storage while moving a particular kind of data. The codes used in characterizing the difficulty of load and store operations have the following meaning:

OK means that there is no overhead or almost no overhead. The memory access is performed using a hardware supported address and only one or two locations are accessed.

OK- means that there is little overhead, but it is more than one extra memory access: it is between three and seven memory accesses.

V means that there is definite overhead of more than seven memory accesses.

OKL means that the operation can be performed without overhead, but only in limited situations. If the programmer wishes to violate these limitations, he must write more complicated code that executes less quickly than the simpler code that would be used if the limitations had not been reached. Slow execution of this complicated code is the source of the overhead for this limited mechanism.

S means that the operation is not hardware supported. Software must interpret the "address" being used to make the memory access or must convert the "pointer" that is being moved from one address space to another.

I/O means that this operation is performed by using disk I/O operations. Often ORSLA or an LAV system would take a page fault instead of performing an explicit I/O operation. I/O operations are much harder for the programmer to use than page faults, but it is not clear how much overhead is involved. The fact that I/O is hard to use suggests that it will often be used inefficiently.

NO means that this operation is seldom supported by this type of system in either hardware or software.

5.4.1 Results of the Comparison

As we inspect this table, we note that wherever there is an OK or an OKL for an SAV system there is also an OK for ORSLA except for accessing external pointers to code (EPC) or permanent storage (EPP) within storage containing machine code (C). Similarly wherever there is an OK for an LAV system there is an OK for ORSLA. The bulk of the overhead for maintaining the lists of inter-area links on ORSLA is invoked when doing operations that are not supported by hardware on SAV or LAV systems. According to the table, the only overhead for frequent operations on ORSLA that is worse than that on SAV systems is the handling of subroutine calls to library subroutines, but the discussion on page 145 suggests that SAV systems may not be better than ORSLA on this point.

It would be possible to reduce the overhead for subroutine calls to library subroutines on ORSLA by using another bit in the *area_information* field of an area: the *lib* bit. An area could be defined to be a LIBRARY area. All LCAs would be automatically cabled to

Table 2. Comparison of Systems - Load and Store Operations

Kind of Storage	Kind of Data	ORSLA	SAV	LAV
T	A	OK	OK	OK
	IP	OK	OK	OK
	EPT	OK	OK	OK
	EPC	OK	OKL	OK
	EPP	OK	OK	OK
ET[11]	A	OK	S[1]	S[1]
	IP	V[4]	NO	NO
	EPT[10]	V[4]	NO	NO
	EPC	OK-[3]	NO	NO
	EPP	OK-[3]	S[1]	S[1,2]
C[8]	A	OK	OKL	OK
	IP	OK	OKL	OK
	EPT[10]	OK[7]	OKL	OK-[5]
	EPC	OK-[6]	OKL	OK-[5]
	EPP	OK-[6]	OKL	OK-[5]
P[9]	A	OK	I/O	OK
	IP	OK	I/O	OK
	EPT[10]	V[4]	S	NO
	EPC	V[4]	I/O	S
	EPP	V[4]	I/O	S

Kinds of Storage

T temporary storage (LCAs)
C areas containing machine code
P permanent data not containing machine code
ET external temporary storage (for another process)

Kinds of Data Accessed

A atomic (no pointers)
IP internal pointers (intra-area pointers)
EPT external pointers pointing at temporary storage
EPC external pointers pointing at machine code
EPP external pointers pointing into permanent storage

Table 2 (con't)
Ratings of the Systems

OK cost for load or store is one or two memory references
OK- cost is more than 2 memory references, but less than 8
OKL cost is only one memory reference, but limited in flexibility
V significant overhead, but not as bad as S or I/O
S access performed by software, possibly I/O
I/O access performed by use of explicit I/O commands
NO not supported or used

Notes

1 SAV and LAV systems perform inter-process communication with special purpose software. The "address" used to perform the access is the name of an event channel or of another process. This must be interpreted by software to actually perform the inter-process communication.
2 Since different processes have different address spaces on LAV systems, the pointers communicated from one process to another cannot be hardware supported pointers but must be software supported pointers such as file names.
3 Since ET storage is reached from the LCA through an inter-area link, any pointers retrieved from ET storage must be checked against cables already made from the LCA. New cables may have to be constructed from the LCA.
4 An inter-area link must be created during this access.
5 Dynamic links are used. Each link must be snapped with software once per process.
6 These pointers are inter-area links, so it is necessary to check that there is a cable from the LCA to this other area. Occasionally, cables must be constructed as well.
7 If SAV systems and LAV systems work at all with EPT pointers, these pointers must point into the temporary storage of the current process. In this case, no check for a cable need be made on ORSLA. The EPT pointer, which uses an inter-area link, will be converted to an IP pointer in temporary storage.
8 Code is usually read and very seldom modified, so the estimates on overhead assume loads rather than stores.
9 Permanent data is often read, but it is also modified. Since the overhead for stores is greater than the overhead for loads on ORSLA, the overhead shown is for stores.
10 Very rare kind of access.
11 Inter-process communication.

---

all LIBRARY areas without having cable objects to each LIBRARY area in each LCA. This would require the system to keep a list of all the LCAs on the system. The memory map would also contain a bit specifying that the area is a LIBRARY area. This would allow calls to library subroutines to be performed without checking for a cable from the

LCA. An EPC pointer from machine code to a library subroutine uses an inter-area link. When this link is used during a call to the library subroutine, it is necessary to obtain the direct reference to the subroutine from the inter-area link and then to obtain a reference to the area containing the library subroutine. At this point, however, the information that this area is a LIBRARY area is obtained, so no further checking is needed since the LCA is always cabled to all LIBRARY areas. Construction of a cable to the LIBRARY area is avoided, as are all the checks that the cable exists. Fortunately, LIBRARY areas are seldom modified and therefore seldom need to be garbage collected. To garbage collect a LIBRARY area, first convert it to a normal area. This is done by constructing cables from all LCAs to this area and then turning off the *lib* bit in the area and in the page map of the pages of the area. After waiting for the LCAs to garbage collect themselves, which should not take more than a few hours or possibly a day since garbage is generated rapidly in LCAs and they are deleted when the processes running in them are terminated, only those LCAs that are still referencing the ex-LIBRARY area will still have cables to it, so it may be garbage collected with no more difficulty than necessary.

Even if this improvement is not made, however, programs originally designed for SAV or LAV systems can be run on ORSLA with very little overhead. There are many rows of Table 2, however, in which ORSLA performs better than SAV or LAV systems. Programs designed specifically for ORSLA will probably run faster than programs designed for SAV or LAV systems that perform the same task, even though lists of inter-area links must be maintained on ORSLA but are not maintained on SAV or LAV systems. Thus, although at first there appears to be some runtime overhead on ORSLA for maintaining the lists of inter-area links, this is only in comparison to an ideal system with a single, large address space on which every load and store can be done with exactly one memory reference. In comparison to real systems, however, there is no runtime overhead for maintaining the lists of inter-area links.

The detailed analysis above ignores an important difference between ORSLA and SAV and LAV systems. When the LCA is garbage collected on ORSLA, another area is created into which all the information in the LCA is copied. The garbage collector, however, runs in the old copy of the LCA and has a cable to the new copy of the LCA.

Store operations into the new copy of the LCA, however, require as many checks as stores into permanent areas. On an SAV or LAV system garbage collection is done within temporary storage. Thus the stores into the new copy of the LCA qualify as stores to P storage on ORSLA but correspond to stores to T storage on SAV and LAV systems. Since storing into P storage is more difficult than storing into T storage, there is additional overhead in garbage collection on ORSLA for maintaining the lists of inter-area links. Thus, although the maintenance of links is done during all computation, it only increases costs during garbage collection. Even then, the cost of constructing links and cables should not really be counted as overhead because the links and cables need to be copied as part of the garbage collection. The real overhead in the garbage collection is the constant checking for inter-area references and checking for cables that already exist. This checking, however, also creates cables that are needed from the new area; thus cables are deleted automatically when an LCA no longer references an area since those cables that are no longer needed will not be created in the new copy of the LCA. Thus the overhead in the garbage collector for maintaining the lists of inter-area links does perform some legitimate garbage collection functions. Furthermore, it is acceptable for the garbage collector to spend time maintaining the lists of inter-area links since the garbage collector benefits enormously from their existence.

Chapter 6
Placement of Objects in Areas

Areas help achieve locality of reference and aid the garbage collector because the objects in an area are all related to each other: they form a logical module of information. It is still necessary, however, to consider mechanisms that cause the objects in an area to be related to each other. Ultimately, the user is responsible for the proper placement of objects, but it is possible for programmers to write programs that correctly place the objects they create. In addition, ORSLA provides as part of the garbage collector a feature known as the *mover* which can move an object that is residing in one area into another area. When used explicitly, it allows the user to move objects from one area to another, but the mover is also able to identify which objects have been placed very poorly and move these objects automatically into more appropriate areas. These mechanisms on ORSLA make it easy for the user to place related objects into the same area.

"Easy" is a relative term, however. I will try to show that placing objects into areas on ORSLA is easier than placing data into files on an SAV system or into segments on Multics. CUID systems, however, do not have a concept similar to areas. Since the programmer on a CUID system does not have to do any placement of objects at all, it is harder to use ORSLA than CUID systems in this respect. Nevertheless, the improved performance of ORSLA over CUID systems is worth the extra effort, especially since mechanisms are supplied on ORSLA that can place objects automatically. When ease of use is important, ORSLA can relieve the user of the need to place objects into areas explicitly, but when efficiency of computation is important, ORSLA can achieve high performance.

6.1 Initial Placement

Every object on ORSLA resides in an area, so when an object is created, the programmer must specify in which area the object should be placed by providing the area as an additional argument to the program that creates the object. This is the initial placement of the object. Although assembly language progreammers and PL/1 programmers are familiar with this argument to the allocate procedure, LISP programmers

and Algol 68 programmers are not familiar with this argument. Fortunately, a good default can be supplied for the area in which to create an object: the local computation area. After all, most objects created by programs are intermediate results in a computation and therefore belong in the local computation area. This corresponds to the default area provided in PL/I. LISP and Algol 68 do not have the concept of areas, thus all of the objects in these languages are allocated from the same pool of storage. Since LISP and Algol 68 deal with files through I/O operations, all of the objects in these languages are within temporary storage. The fact that LISP and Algol 68 force the user to place objects into storage that corresponds to the LCA suggests that the LCA is a good default for placement on ORSLA. Thus a program that creates an object on ORSLA takes the area in which to initially place the object as an optional argument. If no area is specified, the local computation area is used. This default initial placement of objects can be viewed as a form of automatic placement. Thus when a programmer does not specify where an object is to be placed, the system automatically places it into the local computation area.

Not all objects are intermediate results of a computation, however. Objects that are part of the final results of a computation that the user wants to save for some time should be initially placed into the area in which the final results will be saved. The program that performs the overall computation will be used many times by different people, so the area in which to initially place the final results of the computation cannot be a constant in this program; it must be a variable. It will probably not be possible to compute in what area to place the final results from the usual set of arguments to this program, so the program will probably take an additional argument that specifies in which area to place the final results. The program that is in control of the overall computation, however, is often invoked explicitly by the user as a command, so the user will have to specify in which area to place the final results of the computation. The user will only be able to do this if he is aware of the areas already in use and what they contain. One role of the file system is to provide this information.

Although it seems that the only alternatives for initial placement are the local computation area or an area specified by the user, in many instances it is possible for a program to determine the proper area in which to place objects. When a data base is

created on ORSLA, the user must specify in which area to place it. Once a data base has been created, however, the user can make modifications to it without having to specify in which area the data base resides because an object reference to one of the objects in the data base can be used to find what area the object, and therefore the data base, is in. The program making the modifications could then place any new objects into the same area as the rest of the data base. The reference to the area is obtained from the page map for the object that is already part of the data base. Obtaining the reference to the area in this way has problems of protection, but the rule of thumb that protection issues should not interfere with legitimate computation suggests that the protection issues can be solved. Having a reference to an object in an area should not automatically allow a program to place objects into the area. If, however, the program is part of the subsystem that maintains the data base in the area, then it seems reasonable that the program should be able to place objects into the area. Thus the ability on ORSLA to find what area an object is in helps reduce the number of user commands that must specify an area in which to place the final results of the command.

Let us see how a program that controls a computation makes use of the information of where the results of the computation should be placed. The program that controls the computation could use this information directly by first waiting for the final results of the computation to be computed and then copying this information into the area that should hold the final results. This is not the initial placement of this information, however. The initial placement of the objects in the final results can only be affected by causing the programs that initially create these objects to create them in the appropriate area. These objects may be created by very low level programs, however. For example, a multiple precision number may be created by a *multiply* operation. If this number is part of the final results of a larger computation and it is to be initially placed in a permanent area, then the *multiply* operation should be called with an additional argument which is the area in which to create the results of the operation. Thus the proper initial placement of objects may be somewhat difficult to achieve. Programs that attempt it usually require one more argument than programs that do not attempt proper initial placement. Passing these extra arguments requires extra CPU time that should be counted as the cost of properly placing objects when they are created. Furthermore, some algorithms do not allow proper initial

placement because it is not always known whether an object is an intermediate result or a final result when it is created. Algorithms that use successive approximation construct proposed solutions that are used as intermediate results if they are not found to be adequate solutions. Successive approximation is a standard technique in numerical analysis and similar techniques have been developed in graph theory as well. Thus although the proper initial placement of objects seems to be very efficient, in fact it has a cost associated with it and in some cases it is impossible to do. Once these reservations have been noted, however, it should also be mentioned that many algorithms can perform the proper initial placement of objects easily and efficiently. The programmer should decide whether the final results of a program should be placed properly when they are created or should be moved after they have been created.

6.2 Directories

Directories on ORSLA are similar to file directories on SAV systems, but are even more similar to the directories on Multics. Directories on ORSLA have basically three tasks. First, they provide names for the objects that the user manipulates directly. These names are character strings that have been selected by the user. Most names are words or abbreviations that remind the user what the object is and what it is being used for. A particular name can only be associated with one object in one directory, but the same name may be associated with different objects in different directories. Second, since the user must manage areas and must pay for the storage used by areas, directories must be displayed to users in a manner that shows what information is kept in an area and how much storage is used by the area. The user is only concerned about the storage that he must pay for, so there is the concept of an area being *controlled* from a directory. The owner of a directory is responsible for the storage used by the areas controlled from the directory, so ORSLA displays the amount of storage used by the areas controlled from a directory. The directory must supply the user with information about the areas he controls so the user can manage his areas effectively. This section will consider the operations available to the user for managing areas as well as how the directory supplies the information that allows the areas to be managed. The third task of directories on ORSLA is to control access to objects. This task is necessary because a directory is a global data

base that converts character strings into object references. Work on CUID systems has already resulted in techniques for using directories to provide names for objects and to control access to objects. Since controlling access on CUID systems is very similar to controlling access on ORSLA, little will be said here about how a directory controls access to objects.

Figure 29 shows a sample listing of a directory on ORSLA. Each area that is controlled from the directory begins with a line that gives the name of the area and some key information about the area that helps the user manage the area. The following lines give the names of the objects within the area and a description of the type of each object. After all the areas that are controlled from this directory have been listed, then the names

---

list this_dir

```
    area1   3/4, 3/4, 0%
      prog1_source LISP_source_code
      prog1        LISP_object_code

    area2   2/5, 2/7, 5%
      mailbox      mailbox
      cal          appointment_calendar
      this_dir     directory

    misc_obj1      LISP_source_code
    other_dir      directory
    Sam            mailbox
    compile        LISP_object_code
    other_area     area
```

This figure shows the result of the list command typed in the first line of the figure. First the named objects in each of the areas controlled from this directory are listed. The header line for each of the areas has several numbers whose meaning is as follows:

areaname storage used/storage quota, address space used/address space quota, gc_index

After the areas controlled from this directory have been listed, all the named objects in the directory that reside in other areas are listed.

Fig. 29. Listing a Directory

of other objects that are not contained in the areas controlled from this directory are listed along with a description of the type of each of these objects. The description of the type of object is derived from the data type of the object; it does not form part of the name of the object. There will, of course, be options available to the user for only listing some of the objects in a directory.

6.2.1 List of Named Objects

The way a directory is listed is very important for making the user aware of the areas that he is responsible for. By listing the objects within an area immediately under the area, the user becomes aware of the kind of information kept in that area and is reminded of the purpose of the area. It is then easier for the user to choose in which area to place new objects. Listing the named objects in an area is only useful for controlling the area, however, if the objects named in the area describe all of the information in the area. This does not mean that all of the objects in the area must be named, however. What the user considers to be a single object, such as a mailbox, may in fact be a complex data structure that makes use of many objects. The user thinks of it as one object because in order to manipulate the entire structure, it is only necessary for the user to name one of the objects. Object references for the other objects in the structure may be obtained from the named object, from objects accessible from the named object, and so on. Thus a named object will usually be the root of a data structure all of which may be manipulated when the user invokes operations on the named object.

The objects in an area that are named in the directory that controls the area should allow all of the information in the area to be manipulated. In Chapter 4 we saw that the garbage collector needs a more reliable source of accessible objects in an area than is provided by the list of incoming inter-area links. The objects in an area that are named in the directory that controls the area is this reliable source of accessible objects. It should be possible to find all of the objects in the area by starting from each of the named objects and tracing through objects in the area. In order to identify the named objects for the garbage collector, another field is added to the area object: the list of named objects. This is a list of the objects in the area that are named in the directory that controls the area.

This list contains the actual names of the named objects in the area as well as references to these objects. The list of named objects in an area is contained within the area, but the information in this list is duplicated in the directory that controls the area. The directory has a large hash table that contains all the names in the directory so that character string names can be converted quickly to object references for the objects they name. In order to keep these two copies of the information consistent, the user manipulates the list of named objects in an area through the directory that controls the area and the directory corrects both its hash table and the list of named objects within the area.

It should be remembered that a directory does not maintain any sensitive system information on ORSLA and so is just an ordinary object that could have been defined by a user. A directory is a reasonably complex data structure, however, and so will probably consist of several objects. The object reference for the directory refers to the root object of the directory. The objects that comprise a directory must be stored in an area. A directory is a data structure that users will manipulate explicitly with user commands and so the file system must provide a name for a directory. Therefore, the root object of the directory will be on the list of named objects in the area in which it resides. If a directory is placed into one of the areas that this directory controls, then the directory will not need quite as many inter-area links as it will need if it is kept in another area. Thus an unusual aspect of the directories on ORSLA is that one of the names in the directory will probably name the directory itself. Thus the object *this_dir* in *area2* in Figure 29 is the name of the directory that controls *area2*.

The user has the ability to change the names within a directory. Names may be deleted from the directory and names for new objects may be added to the directory. Additional names may even be added to objects that are already named in the directory. A major purpose of a directory is to control a certain set of areas and to display the named objects within these areas. If a directory is controlling a certain area, A, then deleting a name of an object within A from the directory also removes the object from the list of named objects in A. Adding a name for an object within A to the directory adds this object to the list of named objects in A. Adding or deleting names from a directory has no direct effect on the amount of storage used in the areas controlled by the directory,

however. The directory merely contains object references to the objects within those areas. More storage is used in these areas when objects are created in them, not when names are added to the directory. Storage is reclaimed from the areas only when they are garbage collected.

6.2.2 Garbage Collecting Areas

The user on ORSLA is ultimately responsible for garbage collecting areas. Garbage collection is one of the most important management operations that must be performed on an area. Fortunately, the system can help the user determine when each area should be garbage collected. When a directory is listed for the user, the last number on the header line for an area controlled from the directory is *gc_index*, an estimate of the percentage of the storage in the area that would be reclaimed by garbage collection.

How can *gc_index* be computed, however? Garbage can be generated in an area in two ways. An incoming link to an area that is the only accessible reference to the object it points to can be destroyed, thereby making the object the link pointed to inaccessible. In addition, the internal structure of the information in an area can be modified to cause some of the objects in the area to be inaccessible. If the list of named objects in the area actually describes the information in the area, however, then inter-area links to the area should not be pointing to objects that are inaccessible from within the area in which they reside. Thus changes in the number of incoming links to an area should not indicate a change in the amount of garbage in the area. If object references stored within an area have been destroyed, i.e. written over, this may be an indication that the structure of the information is changing and that garbage is being generated. Another indicator that an area needs to be garbage collected is the amount of storage allocated for new objects in the area since the last garbage collection. Even if the new objects did not replace other objects within the area, garbage collecting the area might improve the locality of reference of the area. Finally, if objects have been deleted from the list of named objects in an area, then it is likely that the area contains some inaccessible objects. Thus we see that *gc_index* can be computed from the number of modifications to the area, the amount of storage allocated within the area, and the number of objects deleted from the list of named objects since the

last garbage collection. Unfortunately, it is difficult to estimate how much garbage is generated by each one of these activities. Thus no more will be said in this thesis about how *gc_index* is computed. The exact algorithm for computing *gc_index* is an important research topic for future research. Different algorithms may even be needed for different areas or for different applications.

One obvious way to use *gc_index* is for the user to watch it and use it as a guide for invoking the garbage collector explicitly. When should garbage collection occur, however? There are two factors that must be balanced in deciding when to garbage collect an area. First, garbage collection can reclaim storage. If an area contains a great deal of garbage then it should be collected. Second, a small amount of garbage can have a large cost if it is retained for a long time. Thus, if a series of modifications are being made to an area after which no modifications will be made for a long time, then the area should be garbage collected after the last modification has been made. Similarly, if garbage is being generated in an area at a great rate, it is advantageous to wait until the rate of garbage generation has subsided before garbage collecting the area unless the amount of garbage in the area is producing a serious drain on available storage or is increasing the working set of the computation dangerously close to the size of high speed memory on the system. Thus, unless the amount of garbage in an area is excessive, an area should not be garbage collected until after the rate of garbage creation has decreased. In many instances, it will be well to wait until a user logs out of the system and then to garbage collect the areas that the user modified as a background job on the system.

This analysis leads to three mechanisms for automatically invoking the garbage collector. First, when a user logs out of ORSLA, each of the areas controlled by that user is inspected. If an area's *gc_index* is above a threshold level selected by the user, then that area is scheduled for garbage collection as a background job on ORSLA. ORSLA chooses which areas to garbage collect first so as to make the best use of spare CPU time for garbage collection. An area is only garbage collected as a background job when no LCAs are cabled to the area. Second, if the storage quota on an area is exceeded and if *gc_index* is above a certain threshold, then the area is garbage collected and execution is resumed without reporting the storage quota overflow to the user. This mechanism does not prevent

the user from ever getting a storage quota overflow, however. Immediately after such a garbage collection, *gc_index* will be zero. If the garbage collection did not reclaim much storage, then the next time the quota is exceeded, *gc_index* will not be above the threshold used to trigger garbage collection and so the storage quota overflow will be reported to the user. The third mechanism for automatically invoking the garbage collector is used when the working set of a computation exceeds a certain threshold, causing the computation to receive an interrupt. The process may then decide to garbage collect its LCA and it may also garbage collect an area that the LCA is cabled to and whose *gc_index* is above a rather high threshold.

The reader may have gotten the impression that the user will seldom have to specify when garbage collection should occur. Although this may be true, I have not presented the mechanisms in enough detail to allow anyone to estimate how well they will work. The most sensitive issue is the accuracy of *gc_index* as an estimate of how much an area needs to be garbage collected. Analyzing the quality of a specific algorithm for computing *gc_index* can probably be done only on a specific ORSLA system. Thus, until the merit of automatic invocation of garbage collection can be shown, we should expect that the user will bear responsibility for invoking the garbage collector on permanent areas. Multics LISP, however, has shown the practicality of automatic invocation of the garbage collector for the LCA. Multics LISP invokes the garbage collector when the amount of storage allocated since the last garbage collection exceeds the amount of storage in accessible objects that was found during the last garbage collection.

6.2.3 Deleting Areas

Garbage collecting areas, setting quotas on areas, and naming the objects in areas are not the only operations that must be performed on areas, however. There is also the matter of creating and destroying areas. Areas can be created rather easily, but how are they destroyed? One technique would be to wait until all incoming links and cables have been deleted. The system could easily recognize when the last link or cable has been deleted and then simply free all the storage and address space for the area. No garbage collection need be done because there are no references anywhere else in the system to the objects in this

area. The last link to the area would not be deleted until after the area has been deleted from all the directories on the system, however. Since the user who creates an area is charged for its storage, many users will not want to delete an area from the directory that controls the area without having a good idea of when the storage used by the area will be reclaimed. Furthermore, it would be possible for two areas to be completely inaccessible but for each to have an inter-area link or a cable to the other area. Since these areas are inaccessible, they will never be garbage collected (the user cannot initiate the garbage collector) and the system will never notice that they are inaccessible. Thus the technique just described for deleting an area cannot be depended upon. The file system must be responsible for keeping all existing areas accessible from the file system. Furthermore, the file system should ensure that every area that exists on ORSLA should be controlled from an accessible directory. Thus no area should ever be without incoming links or cables. The technique for deleting an area described in this paragraph can be used as a check that the file system is operating properly. If an area is ever deleted because there are no incoming links or cables, then there is a bug in the file system.

Usually a user will not delete an area from the directory that controls the area unless there is no further use for the area or the information contained within the area. There are two ways to delete an area without causing dangling references to be a problem. The first method ensures that all the information within the area is destroyed and so is called a *hard delete.*

Hard delete: To delete area A, first find all incoming inter-area links to objects in A, change them so they reference the *deleted* object and then remove them from the lists of inter-area links. Almost any computation with the *deleted* object causes an error. A reference to the *deleted* object does not contain an address. Change all the inter-area links from A to other areas to reference the *deleted* object and then remove them from their lists of inter-area links. The set of areas that A has cables to, C_o, is then formed. The set of areas that are cabled to A, C_i, is also formed. Cables are added from each area in C_i to each area in C_o whenever these cables do not exist already. Then all the cables from A to areas in C_o are deleted. All the storage associated with the address space assigned to A is returned to ORSLA. This

destroys all of the object references inside of A and causes the null list of outgoing cables to be accurate. The transitive *cabled* relation for all areas other than A has been preserved even though all the cables from A have been deleted. The only remaining references to the address space associated with A are from areas that are cabled to A. The address space associated with A cannot be reused until all the areas that are cabled to A have been destroyed and the cables to A removed. This can be speeded up by causing these areas to be garbage collected. When an area is garbage collected, any references to objects in A will be converted to references to the *deleted* object in the new copy of the area. When the garbage collection is finished, the old copy of the area will be destroyed thereby removing its cable to A. The new copy of the area is not cabled to A because it does not contain any direct references into A. Whenever a dangling reference into A is used to try to access storage, the fact that there is no storage associated with the address accessed causes an error that is similar to the kind of fault caused by the use of the *deleted* object. Thus the dangling references into A will behave like references to the *deleted* object until they are actually replaced by references to the *deleted* object. When the last cable to A is removed, the address space for A is freed and can be reused since there are no longer any dangling references to it. Thus a *hard delete* temporarily creates some dangling references, but it does not allow them to become a problem.

A hard delete actively destroys all the information in the area. Although this will sometimes be desired, there will be other times when the user merely wants to delete all the named objects within the area from the directory and wants to stop paying for the storage in the area. If other users still need some of the information in the area and are willing to pay for it, then the needed information should be saved. This can be done by performing a *soft delete* of the area.

A soft delete begins by deleting the area and all the named objects within the area from the directory that controls the area. Instead of replacing all references to objects in the area from other parts of the system with a reference to the *deleted* object, the area is garbage collected without creating a new copy of the area. An object in the area that is referenced from elsewhere in ORSLA is moved to one of the areas that points to it. Only

the information that is not moved elsewhere is lost.

Soft delete: To delete area A, initiate a garbage collection involving area A and all areas cabled to A. Do not create a new area A' for the information in A. First garbage collect the areas cabled to A. When a reference to an unmarked object in A is found from area B, copy the object into area B' and mark from the object as if it were in area B. When the garbage collection of the areas cabled to A has been completed, mark from the incoming inter-area links to A. When a link from area C is processed, place any unmarked objects found from that link into C. When this garbage collection is over, there will be no inter-area links to A and no areas will be cabled to A, so a hard delete may be performed on A. Area A will be deleted as part of the clean-up of the garbage collection that will also delete the old copies of the other areas that were garbage collected together with A.

As described, the soft delete algorithm forcibly moves information from A to areas that reference that information. The owners of some areas may not want information moved to their area, however. They could make their wishes known by setting a flag in the inter-area link or cable to the area that is being deleted. If the individual links or cables can enable or disable the saving of information from a deleted area, then the soft delete will probably become the most popular way to delete an area. The philosophy behind the soft delete is entirely consistent with garbage collection: objects should only be deleted when everyone is finished with them, not merely when the entity that created them is finished with them. There is no need for this philosophy to require the user who creates an object to pay for its storage during its entire lifetime, however.

6.3 The Mover

Even when a programmer makes an effort to place an object into the correct area, it is still possible for the structure of the data to change, so that after awhile, the object may become part of the information in another area. Thus, it must be possible to move an object from one area to another. This ability is especially important if the original placement of the object was not done with care. Although the system does not know anything about an object when it is first created, after the object has existed for awhile,

references to the object will be stored in other objects that are related to this object. Thus there is a potential for automatically moving an object to an appropriate area after it has been created. An automatic mover may be able to relieve the user and the programmer of much of the work needed to place objects into appropriate areas and to keep them in appropriate areas.

The only mechanism on ORSLA for moving objects in the address space is the garbage collector. The garbage collector finds all of the object references to the object being moved and modifies these references so that they point to the object's new location. Since the garbage collector looks at all the references to the object on the system, the garbage collector might be able to use the location of the references to the object to determine when an object should be moved from one area to another. Earlier we saw that the soft delete algorithm caused objects to move from one area to another. Perhaps the techniques used there can be generalized to a full automatic mover.

A word of caution is needed here. Unless an automatic mover never makes mistakes, there will be some users who will want to specify where their objects should be placed. The automatic mover should not move these objects unless it is sure that the new placement will be superior.

What criteria should be used to move an object? Each area contains a list of named objects that is supposed to describe the information in the area. Thus all of the objects in the area should be accessible by tracing through objects in the area starting from the list of named objects. Objects that cannot be found by tracing from the list of named objects are not part of the information that is supposed to be in this area. These objects are quite likely garbage. If one of these objects is referenced from other areas, it should be moved to one of the areas that references it. This is the only criterion for automatically moving objects that I advocate. It is a reasonably conservative criterion for automatic movement. Any programmer that wants an object kept in a particular area can put the object onto that area's list of named objects.

Thus it appears that the garbage collector can help decide which objects should be moved and where they should be moved to. This is called the *automatic mover* even though

it is difficult to separate it from the rest of the garbage collector. It is also necessary, however, for a programmer to be able to specify that an object is to be moved to a specific area. Thus a *manual mover* is needed as well as the automatic mover. The manual mover will also be integrated into the garbage collector. A garbage collection should not be initiated whenever a single object is to be moved, however. Instead, the requests to move objects should be saved in a list associated with the area in which the objects currently exist. When the next garbage collection occurs, the movement will actually occur. Thus many individual requests for movement may be combined together and may also be merged with automatic movement and garbage collection. Manual movement is thus quite inexpensive.

6.4 Garbage Collection Revisited

This section describes the garbage collector actually used on ORSLA. It implements the automatic mover and the manual mover as well as performing garbage collection This garbage collector is very similar to the garbage collector described in Chapter 4. Garbage collection begins when the area to be garbage collected, A, is passed as an argument to the garbage collector. Area A is placed in the set of areas, S, that will be garbage collected together. All of the areas that are cabled to A are also placed in S. All of the processes that are executing in the areas in S are stopped and placed on a list, P. The areas in S are locked so that other processes cannot begin accessing the areas in S during the garbage collection. A new set of areas, S', is then produced. For each area, X, in S, there will be a corresponding area X' in S'. A mark data base, M_X, is created in the LCA for each area X. The *explicit* cables from X to other areas are copied into X' and are marked as *unused*.

The area A is the first area to be garbage collected. First, however, it is necessary to process the list of objects that are being manually moved from A. For each object, m, that is being moved from A, a new copy m' is made in the area it is being moved to unless that area is in S, in which case the object is placed in the corresponding area in S'. The information in m is not copied into m' at this time, but m's new location is entered in the external mark data base, M_A.

We are now ready to begin the collection of the information in A. Each of the objects

on the list of named objects in A is *collect*ed in the manner described in Chapter 4. If A is an LCA, then any of the processes in P that are in A are *collect*ed as well. Since the list of named objects is supposed to describe the information in A except for temporary information which should be accessible from any process running in A, the only unmarked accessible information left in A should be moved to other areas. In order to achieve this, a *garbage_collection_state* is added to the miscellaneous information in A and this state is now set to *marking_complete*.

We are now ready to complete the actions required by the manual mover. The list of objects to be moved manually is now processed again. For each object, *m*, a copy, *m'*, has already been made in the required area, but the information in *m* has not been copied into *m'*. We are now ready to copy the information in *m* into *m'*. Any object references in *m* must be *collect*ed and their new versions placed in *m'*. Inter-area links will automatically be generated whenever necessary when an inter-area reference is stored in *m'*. Any unmarked objects that are referenced from *m* should be copied into the area containing *m'*. This automatic movement of objects that no longer belong in the original area makes the manual mover much easier to use than it would otherwise be.

The list of incoming links to A must now be processed. As was noted in Chapter 4, processing the list of incoming links is a sensitive operation because it should not cause cables to be created from the LCA in which the garbage collector is running to any of the areas that have links to areas in *S*. Each incoming link to A is now considered. The links from areas in *S* are not processed at this time because they may no longer be accessible. They are marked as being *possibly_unnecessary*, however, so that they will be processed when it is later determined that they are, in fact, accessible. Each link from an area, Z, outside of *S* (possibly in *S'*) to A is processed by calling *collect_in_new_area* of the object it is linked to, but in such a way that unmarked objects will be copied into Z. If Z is involved in another garbage collection, however, then unmarked objects are copied into A' instead. These complications can be handled by adding two arguments to the *collect_in_new_area* procedure: *previous_area*, and the boolean *may_copy_into_previous_area*. If an area is in the *marking_complete* state, then it will copy any unmarked objects into *previous_area*, that is, the area containing the incoming link, as long as *may_copy_into_previous_area* is true. The

```
procedure collect_in_new_area(y, previous_area, may_copy_into_previous_area);
    begin object y, M; area previous_area, A, N; boolean may_copy_into_previous_area;
    procedure collect(x);
        begin object x, new_x; integer i;
            if externally_marked?(x, M) then return new_copy(x, M);
            new_x := allocate_new_copy(x, N);
            externally_mark(x, new_x, M);
            for i := 1 to size(x) do
                begin object z;
                    GC_load(z, x[i]);
                    begin object w;
                        if storage_monitor?(z) then w := deactivate_monitor(z);
                                    else w := z;
                        if ~atomic?(w) then
                          if IAL?(w) then w := create_new_link(w, N);
                          else if area(w) = area(x)
                              then w := collect(w);
                              else if garbage_collecting?(area(w))
                                    then w := collect_in_new_area(w, N, true);
                                    else ;
                        if storage_monitor?(z) then set_storage_monitor(new_x[i], w)
                              else new_x[i] := w;
                    end
                end
            return new_x;
        end
    A := area(y);
    M := mark_data_base(A);
    if marking_complete?(A) ∧ may_copy_into_previous_area
        then N := previous_area;
        else N := corresponding_area(A);
    return collect(y);
end
```

Fig. 30. COLLECT_IN_NEW_AREA Procedure

---

new code for *collect_in_new_area* is shown in Figure 30. The only change to *collect*, the internal procedure in *collect_in_new_area*, is the recursive call to *collect_in_new_area*.

When the object referenced by an incoming link from area Z has been *collected*, the new reference is stored into the link and the link is unthreaded from the list of incoming links to A and is added to the list of incoming links to the area that the link now points

into. Most of the time the object being referenced will already have been copied into A' by the time this link is processed. If the object was moved by the mover, however, the object may now reside in Z or there may be a cable from Z to the area the object now resides in that makes this inter-area link unnecessary. Thus the cables from Z are checked and if the link is unnecessary, it is marked as *possibly_unnecessary*. The link cannot be replaced by a direct reference at this point because although we know where the link is, we don't know where the reference to the link is. Since the link is now marked as *possibly_unnecessary*, the inter-area link will be replaced by a direct reference the next time area Z is garbage collected.

After the list of incoming links to A has been processed, all of the sources of accessible objects in A have been exhausted except for the accessible links from other areas in S and direct references from other areas in S. These can only be found by garbage collecting other areas in S, however. The process described for A is now followed for the other areas in S. Any direct references from other areas in S to unmarked objects in A will cause those objects to be moved from A because area A is in the *marking_complete* state. All of the incoming inter-area links to A from other areas in S have been marked as *possibly_unnecessary*, so if any of these links are found to be accessible, they will be treated the same as direct references to A. Thus, after all the other areas in S have been processed, all of the accessible objects in the areas in S will have been found and moved to areas in S' or to other areas. The cables from areas in S' to all other areas are now checked to see if any are still *unused*. If so, they are deleted. Remember that an *unused* cable may be deleted at any time, but no other cables may be deleted except by a *hard delete* of the entire area that the cable is from. We may now unlock the areas in S' and resume the processes on the list P. A *hard delete* is performed on all the areas in S and the new copy of the activation record that called the garbage collector is returned to. The garbage collection is now complete.

A garbage collection issue that remains to be explained is exactly what *GC_load* does. *GC_load* is the operation used to read information out of an old copy of an object in order to copy it into the new copy of the object. The hardware described in Chapter 5 is ideal for performing at very high speed the checks ($area(x) = area(w)$) and

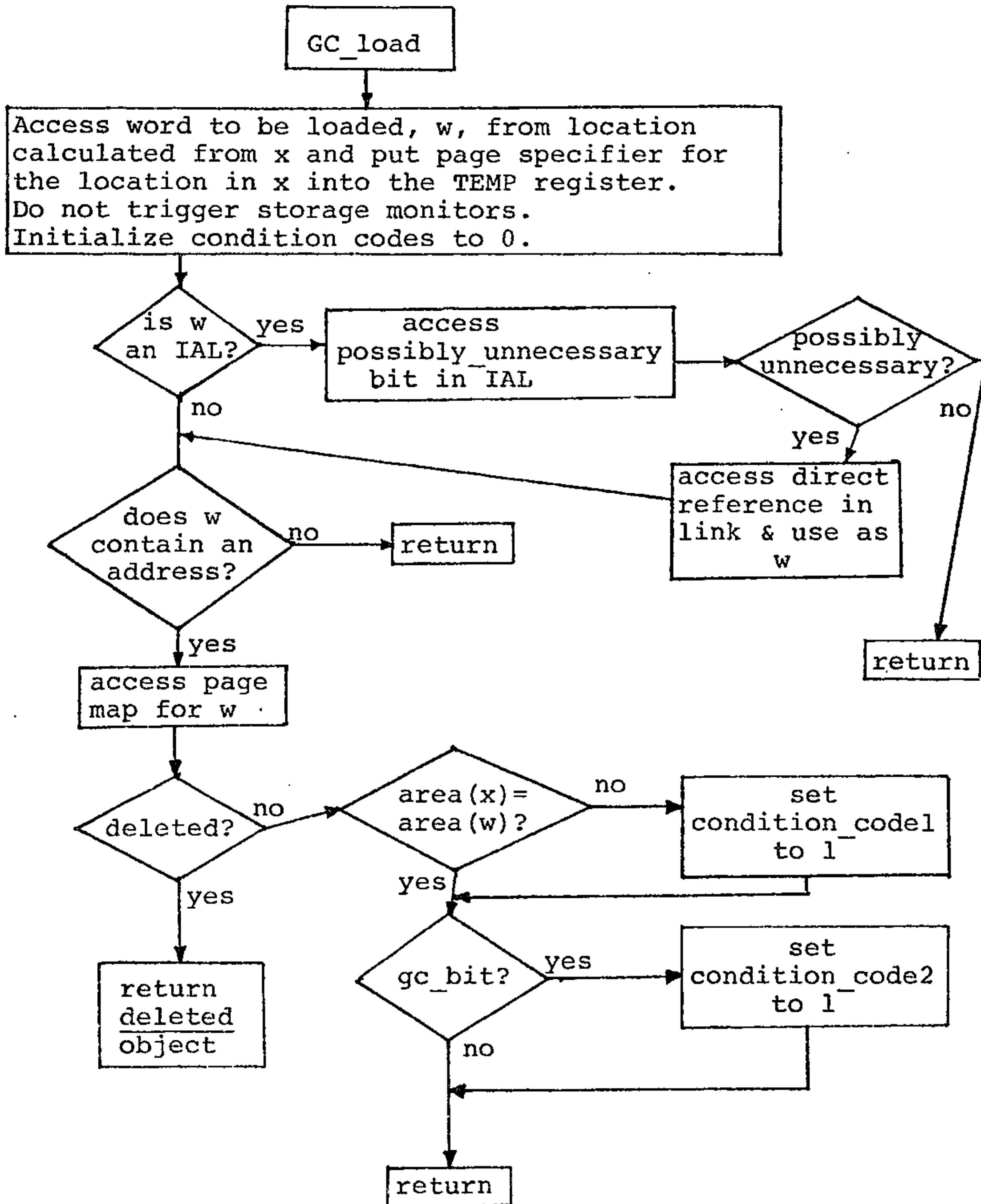


Fig. 31. GC_LOAD Instruction

garbage_collecting?(area(w)) that are required in the *collect* procedure. These checks could easily be performed by the *GC_load* instruction and the results could be returned in the condition codes[1]. I assume that the system has a two bit condition code consisting of the individual bits *condition_code1* and *condition_code2*. We have already seen some of the special properties of the *GC_load* instruction: 1) it does not trip storage monitors, but allows them to be moved into the new copy of the object, 2) it does not automatically return the reference within an inter-area link unless it is marked as *possibly_unnecessary*, thus the garbage collector can stop marking at inter-area links, 3) *GC_load* trips reference monitors so that reference counts will be maintained correctly.

A flowchart of the *GC_load* instruction is shown in Figure 31. *GC_load(x)* begins by obtaining *area(x)* while it loads *w* into the necessary register. *Area(x)* is temporarily placed into the TEMP register in the page map. The two checks (*area(x)* = *area(w)*) and *garbage_collecting?(area(w))* can now be easily computed by accessing the page map using the address contained in *w*, if any. The hardware will then compare *area(x)* (TEMP register) to *area(w)* and will also have the *GC* bit from the page map that indicates *garbage_collecting?(area(w))*. Thus by adding a page map access to *GC_load*, it can return with a condition code that indicates the results of these two tests. The other tests all involve looking at the type code of *w* and so are very fast. This modification to *GC_load* significantly speeds up the most common cases: intra-area references, and direct references into areas outside of the garbage collection. Both of these cases are particularly common when garbage collecting a single LCA. *GC_load* can also use the page map of the object referenced to detect dangling references to deleted areas. The page map entry for the object will contain a *deleted* bit that indicates that the area associated with this page of address space has been deleted. If the *deleted* bit is on, *GC_load* returns a reference to the *deleted* object.

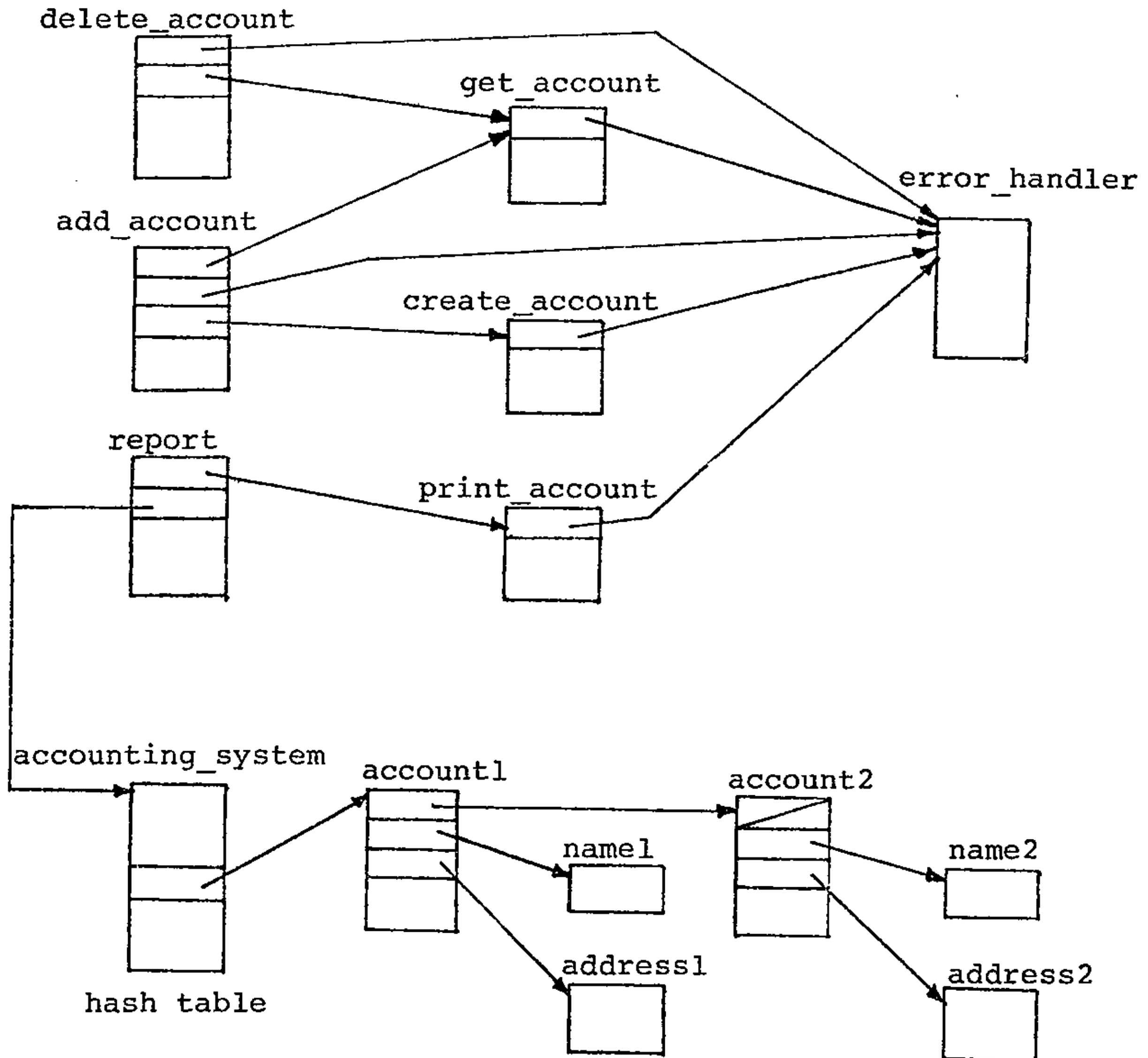
---

1. Some machine languages do not have condition codes but use skips instead. Whatever technique is used to return the results of *compare* instructions should be used by the *GC_load* instruction to return the results of these tests.

6.5 Problems Solved by the Automatic Mover

A serious problem with the placement of objects arises on ORSLA because objects are so small that the user does not want to be aware of most of them. The user wants to be able to deal with large groups of objects as if they were a unit and without knowing how many objects are involved. The first aspect of this problem appears as the difficulty that programs have of deciding where to put their results. Most results are intermediate results in a computation and so should be placed in the local computation area, but some results are final results and so should be placed in other areas. The automatic mover solves this problem by allowing such programs to create their results in the LCA. If they are final results that should be saved, then a reference to them will be stored in the area in which they belong and then the results will become inaccessible from the LCA in which they reside. The computation in the LCA will probably still be able to use the results, but only by accessing the object reference stored in the permanent area in which the results belong. The next time the LCA is garbage collected, the objects that do not belong in the LCA will be moved automatically to the area in which they do belong. Since the length of time between the creation of the objects and the garbage collection of the LCA will be relatively short, there will probably be only one area on the system with references to the objects that were initially, but improperly, placed in the LCA. Thus the automatic mover will not have to choose between several possible areas. Even if the mover does have a choice of areas, however, it just selects any of the possible areas at random. If it later becomes obvious that the objects have been misplaced, then the mover can always move the objects again. Thus the automatic mover allows many programs to be written without concern for where their results are placed. This is particularly important for very general, low level programs such as the *multiply* operation.

Many systems require the user to be aware of all of the objects known to the operating system: usually files and jobs. On such systems objects are so large that these objects usually correspond to objects that the user wants to manipulate directly and so wants to be aware of. On ORSLA, however, a file is a complex data structure consisting of perhaps thousands of objects. The automatic mover allows the user to be unaware of most of these objects. The user need only be aware of the objects he will manipulate directly:



This figure shows two separate modules of information that are somewhat related. The lower module is the data base for an accounting system. Only a few of the objects in this data base are actually shown. The upper module is the machine code of the accounting system software, consisting of seven procedures. The four right-most procedures are internal interfaces in the accounting subsystem, while the three left-most procedures are entry points into the subsystem. Only these three procedures need be named in the area that contains the accounting subsystem software, while only the hash table in the data base need be named in the area that contains the data base. Note that the *report* procedure contains a reference to the data base because a report on this particular data base is made if *report* is called with no arguments.

Fig. 32.

those objects that the user wants to have names for. Many complex data bases need a name for only one of the objects in the data base: the root of the data base (see Figure 32). Such a data base forms a module of information that is always used by starting at the root of the data base. Once the root of such a data base has been placed onto the list of named objects in an area, the automatic mover has a good chance of identifying the rest of the data structure and moving it all into the area that contains the root object. The only references to the internal objects of the data base should be from temporary storage in LCAs that are using the data base and from other objects in the data base. The automatic mover usually gives precedence to inter-area links rather than direct references when moving an object, so internal objects in the data base should migrate to the area containing the root object since they will be referenced with inter-area links from this area. Other permanent areas will usually contain a reference to the root object of the data base since these references will eventually allow the data base to be used, which occurs starting from the root object.

Some complex data structures consisting of many objects may be used starting at several of the objects in the structure. These objects are the entry points into the structture. The user will need to have names for all the entry points so that the data structure can be used. A common example of a data structure containing several entry points is the code for a subsystem (see Figure 32). All of the entry points into the subsystem must be named so they can be called from outside of the subsystem. Special purpose subroutines used only by this subsystem do not need to be named since they are not used from outside of the subsystem. If all of the entry points to a data structure are placed on the list of named objects of the same area, then the entry points into the data structure will be forced into that area. Inter-area links from other permanent areas into this data structure will only reference entry points into the structure, so the only references to the rest of the data structure will be from the entry points into the structure or from LCAs that happen to be using the data structure. Since the automatic mover gives precedence to inter-area links, the structure will be moved into the area containing the entry points of the data structure rather than into an LCA. Thus we may expect that the user will have to be aware of the placement of all the entry points into a complex data structure but the user should not have to be aware of the objects that make up the structure. Since a data structure is

manipulated through the entry points into the structure, the user will need to have names for the entry points in order to manipulate the structure.

Another problem solved by the automatic mover is that the information in an area may have little to do with the named objects in the area, especially if the user explicitly places objects into the area. The automatic mover ensures that the list of named objects really does describe the information contained in the area. This feature allows the user to control what information is kept in each area by specifying in which area each named object is to be placed. If the user decides that two areas should be combined, all of the named objects in one area can be manually moved to the other. The automatic mover gives precedence to references from objects being manually moved over incoming inter-area links, so when the area being abandoned is destroyed with a *soft delete*, all the information in the area will move into the other area. It is not quite so obvious that an area can be divided so easily, however. If an area actually contains two complex data structures and if all of the entry points of the two structures have been identified and are on the list of named objects, and if all the entry points for one of the structures are manually moved to another area, then it is clear that the entire data structure will be moved automatically into the new area the next time the original area is garbage collected. Thus, given all these preconditions, areas can be divided successfully. If, on the other hand, the two data structures in the area are related more closely to each other than to other data structures, then some of the entry points into the structures may not have been named because they were not used from outside the original area. The automatic mover will not properly divide such structures unless the entry points between the structures are also identified. Parts of the data structure that should be moved will remain in the original area due to the basic conservatism of the automatic mover. It will, of course, be possible to identify the rest of the entry points at a later date and manually move them to the proper area, at which time the rest of the data structure associated with those entry points will be moved automatically to the proper area.

Thus we see that if the user takes the responsibility for identifying and naming all of the entry points into a data structure placed into an area, then the automatic mover will place the rest of the objects in the structure into the same area. If, on the other hand, the

user writes software that initially places objects into the proper area, then the user need not name all of the entry points into an area, it is only necessary for the user to name the roots of the data bases in the area. The difference occurs because in the first case we wanted to be sure that the proper information would be moved into the area regardless of where it was initially placed, while in the second case we are only concerned with preventing information from being moved out of the area. The automatic mover is basically conservative in that it will not move an object to another area if it is still accessible from the area it currently resides in. Thus the automatic mover can be used by those who do not want to take any responsibility for placement of objects other than named objects but can also be used by those who want to take some more responsibility for the placement of objects.

6.5.1 Multiple-Area Cycles

We are now ready to deal with a very serious problem of garbage collecting areas separately. The question is: "Does separate garbage collection of an individual area actually get rid of all the garbage in the area?" The answer to this question is "Yes, if the automatic mover is used as well." Without the automatic mover, a multiple-area cycle using inter-area links can prevent garbage from being reclaimed as we saw in section 4.12. With the automatic mover, however, a multiple-area cycle that is inaccessible from anywhere in the system will be consolidated into fewer areas as the areas that hold parts of the cycle are garbage collected until the entire cycle resides in a single area. Garbage collecting this area will then reclaim the storage for the entire cycle.

Let us examine this claim a little more carefully. Let us consider the inaccessible multi-area cycle C consisting of the objects $c_1, \ldots, c_n$. These objects reside in areas $A_{c1}, \ldots, A_{cm}$ which form a set of areas A_c. Since the objects in C form a cycle, we know that we can start from any object c_i in the cycle and follow object references through other members of the cycle to reach any other member of the cycle. Thus all of the members of the cycle are ultimately accessible from any member of the cycle. Just because the objects in C are inaccessible does not mean that objects do not exist that contain object references to objects in C, however. The set B of inaccessible objects are those objects that are not

members of C but that contain object references to members of C. In addition, inaccessible objects that contain object references to objects in B are also members of B. The cycle C cannot be recognized as inaccessible until all of the members of B have been reclaimed. Note that no members of C contain object references to objects in B. Thus the inaccessible objects in B ultimately point to the inaccessible objects in C. The objects in B reside in areas in the set A_b. Note that no accessible object can contain an object reference to any of the objects in $B \cup C$. Thus the objects in $B \cup C$ contain all of the object references in the whole system to the objects in $B \cup C$. Note that none of the objects in $B \cup C$ are area objects or are on the list of named objects in any area because all of the objects in $B \cup C$ are inaccessible, while area objects are accessible from the file system and all the objects on lists of named objects in areas are accessible from area objects.

In order to simplify things, I will assume that none of the objects in $B \cup C$ have pending requests to be moved to another area. If any requests to the manual mover for objects in $B \cup C$ are pending in an area, X, then area X should be garbage collected. The pending move request will thereby be processed. No additional move requests for objects in $B \cup C$ can be generated because these objects are inaccessible. Another way of looking at this requirement is to consider an object with a pending move request to be accessible to the manual mover. This is an accurate picture because the manual mover assumes that all objects being moved are accessible. Storage for these objects cannot be reclaimed until the areas to which the objects have been moved are garbage collected.

We are now ready to state the claim a little more precisely. The storage for the inaccessible objects in the sets B and C will be reclaimed by the time all of the areas in the set $A_b \cup A_c$ have been garbage collected one or more times. It is not necessary for the areas in $A_b \cup A_c$ to be garbage collected together. In most normal cases it will only be necessary to garbage collect the areas in the set $A_b \cup A_c$ once, but there are some pathological cases that could require some areas to be garbage collected again before the objects in B and C can be reclaimed.

This claim is proved by induction. We already know that if all the areas in $A_b \cup A_c$ are garbage collected at once, the storage for all of the objects in $B \cup C$ will be reclaimed

because there are no object references outside of areas in $A_b \cup A_c$ to objects in $B \cup C$ and none of the objects in $B \cup C$ are accessible. This is the basis of the induction. The induction step states that when a group of areas, A_g, are garbage collected together one or more times, then the objects in B that are in areas in A_g will either be reclaimed or will be moved to areas in A_b-A_g and the objects in C that are in areas in A_g will be moved to areas in $(A_b \cup A_c)$-A_g. Thus after the garbage collections, all of the objects in $B \cup C$ will be in the areas $(A_b \cup A_c)$-A_g. In most normal cases only a single garbage collection of areas in A_g will be necessary, but there are some pathological cases, which will be seen below, that could require the areas in A_g to be garbage collected more than once.

Let us now consider the garbage collection of area G from the set A_g. A new copy of the area is constructed: area G'. The sets B_g and C_g contain the members of B and C respectively that are in area G. The garbage collection of the areas in A_g proceeds by garbage collecting areas in A_g one at a time. Thus some areas in A_g will be garbage collected before area G and some areas in A_g will be garbage collected after area G. The garbage collection of area G itself is broken into two important phases: before and after area G is placed in the *marking_complete* state. Before G is placed in the *marking_complete* state two things occur. First, the objects being manually moved are moved to their new areas. We have assumed that none of the objects in $B \cup C$ are being manually moved. Second, the list of named objects in area G is garbage collected. Since every list of named objects is accessible from the file system, all of the objects processed during this phase of the garbage collection are accessible objects and so do not include objects in $B_g \cup C_g$. After G has been placed in the *marking_complete* state the objects that have been manually moved are marked from and the incoming inter-area links to area G are processed. None of the objects in $B_g \cup C_g$ are accessible from the objects that have been manually moved, so we need only consider the second part of this phase. Incoming links from other areas in A_g are marked as being *possibly_unnecessary* but no other action is taken with these lists. Incoming links from areas outside of A_g are marked from. In many cases the object linked to will already have been moved to area G', but if the object is unmarked, it is moved to the area the link comes from unless that area is involved in another simultaneous garbage collection. During this phase we will process all the inter-area links from areas in $(A_b \cup A_c)$-A_g to objects in the set $B_g \cup C_g$. An object that is linked to directly will be moved

to an area in $(A_b \cup A_c)-A_g$ unless that area is involved in another simultaneous garbage collection, in which case the object will be moved to G'. The object will then be marked from and any unmarked objects found will be moved to the same area the original object was moved to. Thus objects in $B_g \cup C_g$ will be moved to areas in $(A_b \cup A_c)-A_g$ or, in unusual cases, to G'.

The problem of a simultaneous garbage collection preventing objects in $B_g \cup C_g$ from being moved to an area in $(A_b \cup A_c)-A_g$ can be solved by merely performing the garbage collection of the areas in A_g (now A_g') again. During the second garbage collection of G, a different set of areas outside of A_g will be involved in simultaneous garbage collections and so the remaining objects from $B_g \cup C_g$ may be moved out of area G. Actually, it will be rare when an inaccessible object is prevented from being moved to another area for several reasons. First, we realize that simultaneous garbage collections can interfere with each other and so ORSLA is designed with this in mind. Most garbage collections initiated automatically by a computation will involve just a single area: the LCA. There will rarely be any inter-area links into an LCA that point to inaccessible objects. If there is an inter-area link to an object in an LCA it will be a reference that has recently been created by the computation, so the object should not be inaccessible yet. The object will probably be inaccessible to the LCA, however, so the object will be moved out of the LCA, thereby reducing the number of incoming links to the LCA. Hopefully most garbage collections of permanent areas will be performed as background jobs on ORSLA. These background jobs will be run one at a time to reduce the interference from simultaneous garbage collections.

We have now considered an object, x, that was unmarked when an incoming link to it was processed from an area outside of A_g. We will assume that x has been copied into an area in $(A_b \cup A_c)-A_g$. The object x is marked from. Since x may contain references to objects in $B \cup C$, we must consider what happens when x is marked from. If x contains an intra-area reference to an unmarked member of $B_g \cup C_g$ then that object is also copied into the area in $(A_b \cup A_c)-A_g$ that x was copied into. If there is a reference in x to an inter-area link, there are two possibilities depending upon whether the link is marked as *possibly_unnecessary*. If the link is not *possibly_unnecessary*, then the inter-area link is

copied into the area x has been moved to. Otherwise it is treated as a direct inter-area reference. Finally, x may contain a direct inter-area reference. In this case there are three possibilities: 1) the inter-area reference points into an area outside of A_g, 2) the inter-area reference points into an area in A_g that is in the *marking_complete* state, and 3) the inter-area reference points into an area in A_g that is not in the *marking_complete* state. In the first case, the object reference is merely stored into the new copy of x, possibly creating an inter-area link automatically. The object reference does not point to an object in $B_g \cup C_g$, however. In the second case, the *collect_in_new_area* procedure is called and if the object is unmarked, it is placed in the same area as the new copy of x. In the third case, however, when the *collect_in_new_area* procedure is called on the object referenced, y, in area A_{gy}, the object y is copied into the new copy of area A_{gy}: A_{gy}'. Thus we find that the objects referenced from x that are in $B_g \cup C_g$ are usually moved to an area in $(A_b \cup A_c)$-A_g, but sometimes these objects may be moved to a new copy of an area in A_g. I expect this latter case to be rather rare because of the order in which the areas in A_g are processed. An object in $B_g \cup C_g$ can only be moved to an area in A_g' if, when an incoming link into an area in A_g is processed, one object is moved to an area in $(A_b \cup A_c)$-A_g and then an inter-area reference is followed to another area in A_g that is not in the *marking_complete* state. This inter-area reference will be covered by a cable. Although inter-area links that are marked as *possibly_unnecessary* are treated as direct references, either these links are also covered by a cable or were marked as *possibly_unnecessary* when processing the incoming links to the area it points into. This only happens if the area the link points into is in the *marking_complete* state state, but in this case the area the link points to is not in the *marking_complete* state. This problem can be avoided if the cables between areas in A_g are used to determine the order in which the areas are processed so that if there is a cable from area A_{gi} to A_{gj} then area A_{gj} will be processed before area A_{gi}. It is possible to satisfy this criterion if the cables between areas in A_g do not form a cycle. Thus direct inter-area references between areas in A_g will only be found when they point into areas in the *marking_complete* state. If the cables between areas in A_g form a cycle, however, it is not possible to garbage collect all the areas in A_g before all the areas in A_g that are cabled to them. On the other hand, cycles of cables are strongly discouraged on ORSLA and should not occur. Even if a cycle of cables does occur, however, it merely slows down the reclamation of an inaccessible, multi-area cycle, it does not prevent the cycle from being

reclaimed. Note that it was necessary for one object to be moved to $(A_b \cup A_c)-A_g$ before a direct inter-area reference could cause an object in $B_g \cup C_g$ to be copied into an area in A_g. Thus it will not be necessary to garbage collect the areas in A_g more than $|B_g \cup C_g|$ times before moving all the objects in $B_g \cup C_g$ to areas in $(A_b \cup A_c)-A_g$, but it will be unusual if the areas in A_g need to be garbage collected more than once. During the garbage collection of the areas in A_g most of the objects in $B_g \cup C_g$ will be moved to areas in $(A_b \cup A_c)-A_g$. If we assume that the two pathological cases do not occur, then all of the objects in $B_g \cup C_g$ that are ultimately accessible from objects in $(B \cup C)-(B_g \cup C_g)$ will be moved to areas in $(A_b \cup A_c)-A_g$ and the objects that are not moved will be reclaimed. Note that all of the objects in C_g will be moved unless $(B \cup C)-(B_g \cup C_g)$ is empty since all objects of the cycle are ultimately accessible from each of the other objects in $B \cup C$.

We have now completed the induction step of the proof. Garbage collecting areas reduces the set $A_b \cup A_c$ until all of these areas (possibly only one area) are involved in a single garbage collection, when the storage for all of the objects in $B \cup C$ is reclaimed. This proof shows that regardless of how complicated a multiple-area cycle is when it becomes inaccessible, garbage collection with the automatic mover will eventually reclaim the storage for the cycle. This proof deals with the worst case, but it does not give a good idea of what happens in the normal case.

When considering the normal case of multiple-area cycles, we must remember that garbage collection is the primary method of reclaiming storage on ORSLA, and so areas are garbage collected whenever they have a significant amount of garbage in them. It is against this background of continual garbage collection that we should consider multiple-area cycles. Only a multiple-area cycle that has a brief lifetime will not be affected by garbage collection before it becomes inaccessible. A cycle that has such a brief existence, however, will probably be created by a single computation but if so, it will probably be created entirely in a single LCA. A long-term cycle will definitely be affected by the background garbage collection. In order for such a cycle to remain spread over many areas there must be many objects in the cycle that are entry points into the cycle. Furthermore, the cycle must be accessible from many areas in the system because the part of the cycle in each area must be accessible to the information in that area or the background garbage

collection would move that part of the cycle elsewhere. It would be unusual for so many references to disappear quickly from so many different, and thus presumably unrelated, areas. It would be likely for the references to the cycle to disappear one by one, during which time the background garbage collection would collapse the cycle into fewer and fewer areas. The process of concentrating the cycle into a single area begins as soon as only one object in the cycle is referenced from outside the cycle or as soon as all the references to the cycle are from a single area. Once a cycle has been placed into a single area, the mover will keep it in a single area because when the first object in the cycle is marked, the recursive call to the *collect* procedure will cause all the other objects in the cycle to be marked as well and to be copied into the same area the first object was copied into. Then the cycle can only be spread over several areas either by manually moving objects in the cycle or by modifying the cycle. Thus we see that many cycles will probably be contained within a single area but even when multiple-area cycles do occur, they will probably not be inaccessible.

6.6 Problems Created by the Mover

There are two problems created by the automatic mover. First, placement of objects can have an important effect on the locality of reference of the programs that use those objects. By moving objects from areas in which the objects are not referenced to areas in which they are referenced, the automatic mover increases the probability that these objects will be placed on the same page with other objects that will be used at the same time as these objects. Thus the automatic mover uses a very rough heuristic that may produce acceptable locality of reference. If, on the other hand, the locality of reference in a particular area is very important to a user, then the user may have carefully analyzed what objects should be placed within the area to maximize locality of reference. The user may be satisfied with the relative placement of these objects within the area that is achieved by the copying garbage collector but may not want objects moved automatically into or out of the area. The basic conservatism of the mover, combined with the ability of the user to place the root of a data structure onto the list of named objects of an area to force the mover to keep a structure in the area handle this problem reasonably well, but are inadequate in some cases. Suppose area B supports area A by holding the objects that are

referenced from objects in A but are rarely used when using the information in area A. The existence of area B allows the objects in area A to be more concentrated and so have greater locality of reference when the information in B is not used. The information in area B cannot be placed on the list of named objects because then all the information in B would always be accessible. The existence of objects in B should be determined by whether they are accessible from area A. There may be other instances where the basic conservatism of the mover is inadequate as well.

The second problem is one of protection. Each user must pay for the storage in his areas. Furthermore, there is a limit to the amount of storage available on the system. It is possible for an owner of a large data base to manipulate his references to the data base so that the automatic mover will move it into areas belonging to other users. It should be noted, however, that the automatic mover will only move objects into an area that already references them, so such a malicious user will be taking advantage of people who have decided to cooperate with him. Such malicious behavior could be handled by normal societal methods, such as refusing to cooperate further with the malicious person.

Both of these problems can be handled by adding some special mechanisms to ORSLA that control the automatic mover:

1) there could be flags in the area object that indicate that objects cannot be automatically moved into or out of the area
2) there could be a flag in each inter-area link and cable that indicates whether the link or references covered by the cable could cause automatic movement
3) on each area, there could be a list of other areas that objects can or cannot be moved automatically to or from; this list could contain not only specific areas, but could also characterize the type of area, e.g. the areas owned by a certain person or controlled from a certain directory.

These mechanisms create new problems, however. What should be done with an object that is no longer part of the information in the area in which it resides and is accessible from other areas on the system, but which cannot be moved to any of those areas? There are two choices: 1) the object can be kept in its current area, or 2) the object can be deleted. Each of these choices is appropriate in certain circumstances. The user can be given

control of this choice by adding another bit of information to the mechanisms listed above. If the user interferes with the automatic mover because he is placing objects explicitly, then he will probably want objects kept in their current area. This option raises the problem of multiple-area cycles again. In addition, the list of named objects for an area that the mover cannot move objects out of does not necessarily describe the information in the area very well. Since the user is taking responsibility for the placement of objects, however, these problems should be solved by the user. The second option causes the garbage collector to delete objects that are not accessible from their own area, are accessible from other areas, but which may not be moved to any of these areas. In this case the list of named objects in the area continues to describe the information in the area. A multiple-area cycle may be destroyed by this mechanism before it becomes inaccessible, but when it does become inaccessible, the storage for the cycle will be reclaimed. Thus the second option is better if the automatic mover is being prevented from operating due to considerations of protection. When objects are deleted by the garbage collector, all of the references to the objects are modified so they reference the *deleted* object, thus no dangling references are created[1].

6.7 Costs of Garbage Collection

Now that we have seen the full garbage collector on ORSLA and have an idea of the benefits gained by garbage collection, it is time to consider the costs of garbage collection. Unfortunately, it is not easy to analyze these costs meaningfully. As a rough approximation, the cost of a garbage collection is proportional to the amount of storage in

1. Those readers who are beginning to wonder why all of these complications are necessary when explicit deletion of objects has been used for so long are referred to Chapter 2, where I explain why the concept of objects causes garbage collection to be so desirable.

accessible objects found by the garbage collector[2]. A more meaningful measure, however, would be the cost of garbage collection per word of reclaimed storage. If we assume that a garbage collection uses an amount of CPU time that is proportional to the amount of storage in accessible objects and that its only benefit is the amount of storage reclaimed, then it is obvious that the cost in CPU time of reclaiming a word of storage can vary widely depending upon when the garbage collector is invoked.

In order to analyze when to invoke the garbage collector, it will be necessary to make several simplifying assumptions that are not often valid. After I have performed this simplified analysis, however, I will use the results of this analysis to further analyze the cases in which the assumptions of the simple analysis do not hold.

Let us consider garbage collection of the area A that contains n words of accessible objects. If C_{gc} is the cost of garbage collection per word of accessible object, then nC_{gc} is the cost of a garbage collection of area A. Let us assume that the information in area A has a certain rate of garbage generation, r, associated with it. Hence r is the number of words of garbage generated per day per word of accessible object in area A. Thus nr is the number of words of garbage generated per day in area A. The true cost of garbage collection includes the cost of the garbage as well as the cost of garbage collection. The existence of garbage has two costs. First is the cost of the storage to hold the garbage. This cost is proportional to the average number of words of garbage in area A. The second cost of garbage is due to the fact that garbage decreases the locality of reference of computations that use area A. It is difficult to analyze how this cost is related to the

---

2. Some people may want to include the costs of maintaining the lists of inter-area links. These costs cannot be avoided on ORSLA, however. Furthermore, the maintenance of inter-area links allows ORSLA to operate in a single, large address space while making good use of virtual memory. As we saw in Chapter 5, these advantages seemed to allow ORSLA to run faster than other systems, so if we are trying to compare costs of garbage collection between two systems, the loss in speed on these other systems due to poor paging or due to multiple address spaces should be counted against the cost of garbage collection on these systems if maintenance of inter-area links is counted against garbage collection on ORSLA.

number of words of garbage in area A. If a computation that uses A has a large working set and the garbage in A causes the computation to thrash, then the cost of garbage increases more than linearly with the amount of garbage. To handle these uncertainties, I assume that if there are x words of garbage, the cost of this garbage is $x^a C_s$, where a and C_s are parameters of area A. If t is the number of days since the last garbage collection, then nrt is the number of words of garbage currently in area A and this garbage costs $(nrt)^a C_s$ dollars per day. If the amount of storage in accessible objects, n, and the rate of garbage generation, r, are constant, then area A will be in a steady state and there will be a constant frequency of garbage collection, f, in days^{-1}. When it is time for a garbage collection, $f = 1/t$. The total cost per day of garbage and garbage collection, T, for area A is the sum of the average cost per day of garbage collection, T_{gc}, and the average cost per day for garbage, T_s. The average cost per day of garbage collection is just:

$$T_{gc} = nfC_{gc}$$

The average cost per day of garbage is

$$T_s = \frac{1}{t}\int_0^t (nrt)^a C_s dt = f\int_0^{\frac{1}{f}} (nrt)^a C_s dt$$

$$T_s = \frac{C_s}{a+1}\left(\frac{nr}{f}\right)^a$$

Thus

$$T = nfC_{gc} + \frac{C_s}{a+1}\left(\frac{nr}{f}\right)^a$$

We choose f by trying to minimize the total cost per day of garbage and garbage collection, T.

$$\frac{dT}{df} = nC_{gc} - \frac{a}{a+1}\ \frac{C_s (nr)^a}{f^{a+1}}$$

Setting dT/df to zero and solving for f, we find

$$f = \sqrt[a+1]{\frac{a}{a+1}\ \frac{n^{a-1} r^a C_s}{C_{gc}}}$$

Substituting in the equation for T, we find

$$T_{gc} = n \sqrt[a+1]{\frac{a}{a+1} n^{a-1} r^a C_s C_{gc}^a}$$

$$T_s = \frac{n}{a} \sqrt[a+1]{\frac{a}{a+1} n^{a-1} r^a C_s C_{gc}^a}$$

$$T = \left(\frac{a+1}{a}\right) n \sqrt[a+1]{\frac{a}{a+1} n^{a-1} r^a C_s C_{gc}^a}$$

Taking the log of both sides, we find

$$\ln T = \frac{a}{a+1} \ln \frac{a+1}{a} + \frac{1}{a+1} \left[\ln(n^a C_s) + a \ln (nrC_{gc})\right]$$

Thus T is the weighted logarithmic average of the cost per day of n words of garbage ($n^a C_s$) and the cost of garbage collecting the garbage generated in area A in one day (nrC_{gc}). The reason the cost of n words of garbage is important is that the cost per day of the garbage that exists when we garbage collect, $(nr/f)^a C_s$, is between one and two times the average cost per day for the garbage collection, which is proportional to the number of words of accessible objects in area A.

$$\left(\frac{nr}{f}\right)^a C_s = \left(\frac{a+1}{a}\right) n \sqrt[a+1]{\frac{a}{a+1} n^{a-1} r^a C_s C_{gc}^a}$$

$$= \frac{a+1}{a} T_{gc}$$

Although this analysis reflects the uncertainty of the cost of garbage, it is very likely that the cost of garbage will be proportional to the number of words of garbage, i.e. $a = 1$. In this case we find that

$$T_{gc} = n\sqrt{\frac{1}{2}\, rC_sC_{gc}}$$

$$T_s = n\sqrt{\frac{1}{2}\, rC_sC_{gc}}$$

$$T = 2n\sqrt{\frac{1}{2}\, rC_sC_{gc}}$$

The cost of garbage collection per word of garbage is the average cost of garbage collection per day, T, divided by the number of words of garbage generated per day, nr.

$$\frac{T}{nr} = \sqrt{\frac{2C_sC_{gc}}{r}}$$

Thus we have found that if $a=1$, then the cost of garbage collection is proportional to the number of words of garbage reclaimed, but the constant of proportionality varies from area to area. Those areas with the highest rate of garbage generation will have the lowest cost of garbage collection per word of garbage. The cost of garbage collection per word of garbage is not related to the size of an area as long as all of the information in the area has the same rate of garbage generation.

The cost of garbage collection can be reduced by separating objects with different rates of garbage generation into different areas. Consider the sets of objects, S_1 and S_2 consisting of n_1 and n_2 words of accessible objects and having a rate of garbage generation of r_1 and r_2 respectively. If these objects are separated into their own areas, the total cost of garbage collection for these two areas, T_1 and T_2 is:

$$T_1 = \sqrt{2n_1^2 r_1 C_s C_{gc}}$$

$$T_2 = \sqrt{2n_2^2 r_2 C_s C_{gc}}$$

If these objects are combined into one area, the total cost is

$$T_3 = \sqrt{2(n_1+n_2)^2\left(\frac{n_1r_1+n_2r_2}{n_1+n_2}\right)C_sC_{gc}}$$

$$T_3 = \sqrt{2(n_1^2r_1+n_1n_2(r_1+r_2)+n_2^2r_2)C_sC_{gc}}$$

The ratio of these two costs is:

$$\frac{T_1+T_2}{T_3} = \frac{\sqrt{n_1^2r_1} + \sqrt{n_2^2r_2}}{\sqrt{n_1^2r_1+n_1n_2(r_1+r_2)+n_2^2r_2}}$$

$$\left(\frac{T_1+T_2}{T_3}\right)^2 = \frac{n_1^2r_1+n_2^2r_2+2n_1n_2\sqrt{r_1r_2}}{n_1^2r_1+n_1n_2(r_1+r_2)+n_2^2r_2}$$

$$= 1 + \frac{2n_1n_2\sqrt{r_1r_2} - n_1n_2(r_1+r_2)}{n_1^2r_1+n_1n_2(r_1+r_2)+n_2^2r_2}$$

Separating the objects is better than combining them if $T_1+T_2 < T_3$, i.e. if $2\sqrt{r_1r_2} - (r_1+r_2)$ is negative. Substituting $xr_1=r_2$, we get

$$f(x) = 2\sqrt{xr_1^2} - r_1(x+1)$$

$$f(1) = 0$$

$$f'(x) = \frac{r_1}{\sqrt{x}} - r_1$$

$$f'(1) = 0 \qquad f'(x) > 0 \text{ for } 0<x<1$$

$$f'(x) < 0 \text{ for } x>1$$

Thus $f(x)$ is always negative except when $x=1$, when it is zero. Thus it is better to separate

the objects unless they have the same rate of garbage generation. The basic reason for this is that each set of objects has its own optimal frequency of garbage collection. Time for garbage collection is wasted if a set of objects is garbage collected too frequently and storage is wasted if a set is not garbage collected frequently enough. These considerations are not so important on ORSLA that they have a major effect on what objects are placed in an area, but these considerations do cause garbage collection on ORSLA to be much more efficient than on systems on which the entire system is garbage collected at once.

Another factor that allows the cost of garbage to be reduced significantly on ORSLA is that the assumption that garbage is generated at a uniform rate is very poor. Permanent areas seem to be modified in bursts, while only a few areas have a constant rate of modification and therefore garbage generation. The fact that files are modified in bursts is used by systems that automatically backup their disk files by creating incremental dumps that contain only the files that have been modified since the last dump. It remains necessary to perform a complete dump occasionally, but if most files on the system were being modified at a uniform rate, there would be no advantage to doing incremental dumps at all. If permanent areas are modified in bursts, then the cost of garbage can be reduced by garbage collecting an area immediately after a burst of garbage generation. There will be no garbage in the area until the next burst of garbage generation and the garbage generated by the burst just before the garbage collection will not have existed for long and so will have been very inexpensive even though it allowed more garbage to be reclaimed during the garbage collection. If the bursts of garbage generation are so large that it is necessary to garbage collect the area during the burst as well as after the burst, then the above analysis can be used to help determine when to garbage collect during the burst; we can consider the rate of garbage generation to have changed during the burst. During the quiescent period the rate of garbage generation drops to zero. When trying to determine whether to garbage collect after a burst, however, the long term average of garbage generation should be used rather than the rate during a quiescent period. If the bursts of garbage generation are small, then the area should not be garbage collected after each one. The above analysis can be used to help determine how many bursts should be passed over before garbage collecting the area. If such an analysis tells us to garbage collect during a quiescent period, the area should be garbage collected at the beginning of

the quiescent period.

Perhaps the highest rate of garbage generation will be found in the LCA. Different computations will have different patterns of garbage generation in their LCAs. The rate of garbage generation will not be constant, but it will probably be very high at all times when compared to permanent areas. Since a computation lasts such a short time, it is probable that the cost of disk for the garbage in the LCA will be rather low. Garbage collection in an LCA will probably be forced by increases in the size of the working set caused by the garbage. If the cost of high speed memory is not enough to cause garbage collection, then the cost of thrashing will be, but if garbage collection is triggered by thrashing, then the exponent a will probably be larger than one. This will increase the cost of garbage collection, but it will also serve as a greater incentive to the programmer to reduce the working set of his computation regardless of the amount of high speed memory on his system.

The total number of words of accessible objects in an LCA is not constant. Thus it is important to try to choose the time of garbage collection so it coincides with a small number of accessible objects. The number of accessible objects almost always reaches a relative minima between invocation of user-level commands, since during a command there are many accessible objects that hold the temporary results of that command. The relative minima of accessible objects between commands mark those times when garbage collection can be performed more efficiently than the neighboring times. The analysis above must be used, however, to select which of these relative minima should be used or to determine that garbage collection needs to be performed before the next relative minima is reached.

Fortunately, the analysis above is not restricted by all the assumptions that were made during its derivation. The assumption of a constant number of accessible objects is not very restrictive if we realize that it only refers to the number of accessible objects over the several opportunities for garbage collection that are being considered. Although the number of accessible objects varies greatly during commands, it is relatively constant at those times between commands. In any case, it is not known in advance whether the number of accessible objects will increase or decrease, so a constant level is a good average

prediction. The assumption in the above analysis of a constant rate of garbage generation is somewhat troublesome. The analysis allows the time of garbage collection to be identified regardless of the values of a or r from the estimated cost of the next garbage collection, the total cost of garbage since the last garbage collection, and the cost per day of the current amount of garbage. These factors may be easier to measure or estimate directly than a, r, C_s, or C_{gc}.

Let t be the time since the last garbage collection and K_{gc} be the estimated cost of the next garbage collection. K_{gc} can be estimated by adjusting the cost of the last garbage collection by the increase or decrease in the number of accessible objects that will be involved in the next garbage collection. Let K_s be the total cost of garbage since the last garbage collection and let g be the cost per day of the current amount of garbage. If we were to garbage collect at time t, the average cost of garbage and garbage collection would be

$$T = T_{gc} + T_s = \frac{K_{gc}}{t} + \frac{K_s}{t}$$

We know that under the assumptions of our previous analysis that

$$T_s = \frac{1}{a+1}\left(\frac{nr}{f}\right)^a C_s$$

But nr/f is the amount of garbage at the time of garbage collection and $(nr/f)^a C_s$ is the cost per day of this amount of garbage. At the time of garbage collection, $(nr/f)^a C_s$ will be equal to g, the cost per day of the current amount of garbage. Thus:

$$T_s = \frac{K_s}{t} = \frac{g}{a+1}$$

Solving for a, we find

$$a = \frac{gt}{K_s} - 1$$

Thus, once we have measured g and K_s, we can calculate a. If there are non-linearities in the cost of garbage, they can be approximated reasonably well by the assumption that $x^a C_s$

is the cost of x words of garbage. If the cost of garbage is actually exponential (due to thrashing), we will find the value we calculate for a increasing with time. To a limited degree, the ability to calculate a from g and K_s means that if the rate of garbage generation is changing, it will affect the value calculated for a.

From the previous analysis we know that at the time of garbage collection:

$$\left(\frac{nr}{f}\right)^a C_s = g = \frac{a+1}{a} T_{gc} = \left(\frac{a+1}{a}\right) \frac{K_{gc}}{t}$$

Substituting for a, we find:

$$g = \frac{K_{gc} + K_s}{t}$$

Thus we should garbage collect the area whenever

$$g \geq \frac{K_{gc} + K_s}{t}$$

This algorithm allows us to handle variations in the number of accessible objects fairly well, but it handles variations in the rate of garbage generation less well in certain circumstances. If, during the period since the last garbage collection, there has been a large increase or decrease in the rate of garbage generation, then a rather large or small value for a will be calculated. The previous analysis assumes a steady-state, however, in which the next cycle between garbage collections is like the current one. That means that the analysis expects the rate of garbage generation to change to the rate at the beginning of the current cycle when the garbage collection is performed. Performing a garbage collection will not affect the rate of garbage generation, however. The effect of $(a+1)/a$ is very large when a is small but is not important when a is large, thus no great harm is done if the value calculated for a is large due to an increase in the rate of garbage generation rather than a non-linearity in the cost of garbage. The cost of garbage is at least proportional to the amount of garbage, so a should not be less than one. I propose limiting

$(a+1)/a$ to a maximum of two to reduce the seriousness of the error if the rate of garbage generation is decreasing. Thus the modified algorithm is that an LCA should be garbage collected when

$$g \geq \frac{K_{gc} + \min(K_s, K_{gc})}{t}$$

This algorithm does not minimize the cost of garbage collection in all cases, but the errors introduced by this algorithm will probably be much less than the errors caused by errors in measuring and estimating g, K_{gc}, and K_s. This algorithm is a good starting point for ORSLA. Future research will be able to refine the algorithm for automatically invoking the garbage collector, but the algorithm given here should ensure that the cost of garbage collection per word of reclaimed storage will be inversely proportional to the square root of the rate of garbage generation.

6.8 Garbage Collection Corresponds to Other Operations on SAV Systems

Some people may have concluded that garbage collection is a major source of overhead on ORSLA. After all, other systems don't do much garbage collection. This issue is not at all clear-cut, however. Garbage collection is performed on ORSLA to handle fundamental problems of storage fragmentation and locality of reference as well as to reclaim storage. It may appear that LAV and SAV systems do not have such problems within their files. I would suggest that this is because these systems currently do operations that are equivalent to garbage collection.

On SAV systems, especially, we note that often a file is modified by performing a pass over the entire file, copying it into high speed memory, processing it, and then writing a new file. I suggest that this corresponds to two operations on ORSLA: 1) making a modification to an area, and 2) garbage collecting the area. Notice that a *new copy* of the file is made, thus corresponding to a *copying* garbage collector. Combining modifications with garbage collection, however, has an important consequence. It becomes very expensive to make a single, small modification to a large file. To eliminate this problem, modifications are collected on SAV systems until there are a large number of them and

then they are all made at once, thus spreading the cost of the "garbage collection" over many modifications.

On ORSLA, however, modifications to a data base are naturally separated from garbage collection. Single modifications are not too expensive so they can be processed immediately. It is still possible to save paging time for code and the data base itself by accumulating modifications and making them all at once, but the desirability of having the data base accurate may outweigh this difference in cost. On ORSLA, the user may wait until the data base contains much garbage before garbage collecting it, rather than being forced to garbage collect it because some modifications must be made.

The desirability of separating modifications from garbage collection has not been ignored on SAV systems, however. Techniques have been developed on these systems for operating in the way I just described for ORSLA. List processing is very helpful in making single modifications, however, while list processing is less necessary if the entire file is regenerated whenever a change is made. SAV systems provide very little support for list processing within files while on ORSLA permanent areas may contain object references that are interpreted by the hardware.

Perhaps the most frequent reason for copying information is to move it between storage devices of different speeds. Virtual memory performs all of this copying (paging) on ORSLA and Multics, but I/O operations do some of this copying on SAV systems. It is well known that virtual memory can decrease copying of information between disk and high speed memory substantially in many instances by taking advantage of the instantaneous state of the computer system. Another important tool to reduce copying on ORSLA is *sharing*. The same copy of a data structure can be used many times by copying an object reference to the data structure instead of copying the data structure itself. Finally, however, it becomes necessary to do some copying of data structures to make them compact. By isolating this copying in a garbage collector, however, it can be invoked only when it is really needed. In fact, the necessity of garbage collection is a consequence of widespread sharing. If objects were not shared, each object would be used by only one program so it would not be necessary to use garbage collection to reclaim storage since storage for an

object could be explicitly freed by the program using the object. It would be necessary to continually copy information from one object to another in order to avoid sharing, however. It may be that garbage collection is the price we must pay for comprehensive sharing of information. Sharing substitutes copying a single object reference for copying a larger structure during normal computation, and enables the speedy modification of data structures through the use of side-effects. The savings gained in these ways may well outweigh the costs of garbage collection on ORSLA.

6.9 Comparison of Inter-area Links with Linking on SAV and LAV Systems

Linking on SAV and LAV systems allows a program, A, to be compiled separately from the programs that use A and the programs called by A. The object code that is produced by a compiler on an SAV or an LAV system is not quite ready to be executed, however, because the compiler does not know where the subroutines called by A are. Part of the object code consists of a list of external symbol references that must be defined before the code can be run. Another part of the object code contains several symbol definitions for the entry points into the object code that may be used by other programs. Usually each external symbol reference consists of a word in which to store the symbol definition and a character string that is the name of the symbol it should be linked to. Such a symbol reference can be viewed as a *link*. When linking occurs, the definition of each link is found and stored in the link. This is known as *snapping* the link. If it ever becomes necessary to unlink, then a zero is stored in the link in order to *unsnap* the link. This is rarely done on SAV or LAV systems but is necessary on SAV systems when a program is purged from the address space to make room for another program. A link is snapped by finding a symbol definition whose character string matches the character string in the link.

Linking is necessary on SAV and LAV systems because of the short lifetime of an address space on these systems. This kind of linking is less necessary on ORSLA because a program can remain linked for long periods of time to the subroutines it calls. Without considering the needs of separate compilation, however, we have needed two mechanisms on ORSLA that are similar to the mechanisms needed to support linking. First, ORSLA

has inter-area links. Inter-area links on ORSLA remain snapped for long periods of time. Second, ORSLA has the list of named objects in an area that corresponds to the symbol definitions in object code. The only mechanism that exists in object code on SAV and LAV systems that does not exist on ORSLA is the presence of the name of the symbol within a link. This mechanism allows links on SAV and LAV systems to be snapped. We have not considered the possibility of snapping an inter-area link on ORSLA. So far we have always created a link by providing the actual object reference rather than merely the name of a symbol. In this chapter we have seen that inter-area links are unsnapped when the object they point to is deleted with a *hard_delete*. In the next chapter we will see more widespread unsnapping of inter-area links in order to implement revocation of access. If inter-area links are to be unsnapped, however, it will be convenient to be able to resnap them. Thus it is probably a good idea to add symbolic names to inter-area links on ORSLA. It will be necessary to add two words to an inter-area link to incorporate this change as well as the other information that this chapter has already suggested be associated with an inter-area link. The new format of inter-area link will contain five words:

1) the reference to the object linked to
2) the list of incoming inter-area links
3) the list of outgoing inter-area links
4) a character string name of the object linked to
5) miscellaneous information, including whether the object linked to may be moved to the area containing this inter-area link

If, when an inter-area link is used, it is found to contain a reference to the *deleted* object, then a fault could be generated that would try to snap the link. This brings us to the problem of how to find a reference to the object the link is supposed to point to given only the name of the object. On SAV and LAV systems there is usually a series of directories that are searched until the name is found. The first directory to look in is the directory that controls the area containing the link. The list of directories to be searched could be associated with this directory. In some cases, however, the object linked to is not named in any of the directories in the search. Multics allows this case to be handled by storing the pathname of the object in the link. The pathname is a series of names. The first name is applied to the root directory of the system to find another directory, and so

on, until the last name in the series names the object itself. Both of these options could easily be supplied on ORSLA.

Since the purpose of the name in an inter-area link is to allow the link to be resnapped, there is some difficulty on ORSLA in finding out what name should be associated with an inter-area link. It is not a serious matter if no name is associated with a link on ORSLA. Nevertheless, it is possible to provide inter-area links between permanent areas with appropriate names automatically. If an inter-area link between two permanent areas is created automatically by storing an inter-area reference in one of the areas, the chances are good that ORSLA can associate an appropriate name with the link automatically by merely searching the list of named objects of the area the link points into. It was mentioned earlier in this chapter that the user will provide names for all of the objects in an area that are entry points into the information in that area from other areas. The user will provide these names in order to allow the information in the area to be used properly and also to allow the automatic mover to operate properly. Exactly when a simple name should be generated automatically for an inter-area link and when a full pathname should be generated will be left to future research. There is also a question about when a name should be generated automatically for an inter-area link: when the link is created or when it is unsnapped. This question will be left to future research.

Thus we see that inter-area links and the list of named objects are similar to information kept in object code modules on SAV and LAV systems. By adding names to inter-area links on ORSLA we can provide the ability to resnap unsnapped inter-area links and the ability to create inter-area links to objects that do not yet exist. The programmer who creates an inter-area link to an object that does not yet exist must supply the name of the object. Compilers can use this feature to compile programs that call other programs that have not yet been written. Thus the ability to snap unsnapped inter-area links automatically appears to be useful. We will see in the next chapter that it helps reduce the undesirable effects of revocation as well.

Chapter 7
Protection

Protection on ORSLA is implemented by enforcing the restrictions on the use of object references that are inherent in the concept of objects. The low level restrictions on the use of object references require any program that accesses the representation of an object to have an object reference to that object. The low level restrictions allow the program that creates an object to control the initial distribution of references to the object. The high level restrictions allow the software that defines an object to control what operations a program with a particular object reference may perform on the object and how the operations are implemented. Section 3.3 discussed most of the ways in which the low level restrictions on the use of object references are enforced, but section 3.3 did not adequately describe how a program is prevented from accessing storage outside of an object to which the program has a low level reference and on which the program is performing load and store operations. Each load and store operation must check the size of the object. A technique is described in section 7.1 for encoding most of the size information into a 5 - 9 bit *size* field in the object reference and then storing more complete size information elsewhere when necessary.

Section 3.4 discussed the general technique used on ORSLA to enforce the high level restrictions on the use of object references. The object reference contains a *high-low* bit that specifies whether the object reference is a low level object reference, in which case load and store operations may be performed on the representation of the object, or whether the object reference is a high level object reference, in which case only the operations defined by the object's data type definition can be performed on the object. Actually, the only operation that can be immediately performed on a high level object is to convert it to a low level object, i.e. set the *high-low* bit in the object reference to *low*. Section 7.2 describes the operation that exists on ORSLA for setting the *high-low* bit and shows how any program that is not part of the data type definition of an object is prevented from using this operation on this object. In order to ensure that the operation of lowering the *high-low* bit is fast, it will be necessary to describe the format of the object reference for data type definitions, but no further description of the representation of a data type

definition will be given. Discussion of the representation of a data type definition is beyond the scope of this thesis.

The domain is an important concept in the field of protection. A domain is a set of objects for which object references can be obtained by a particular activation of a procedure and the operations that can be performed on these objects with these object references. On some systems, an entire computation is performed in the same domain, while on ORSLA the domain of execution changes frequently. Domains on ORSLA are defined in a similar way to domains on HYDRA, consisting of all the objects that can be reached from the current procedure activation record. A new domain is entered on every procedure call, but procedure calls on ORSLA correspond to procedure calls in a programming language while procedure calls on HYDRA are inter-subsystem calls. Section 7.3 describes how a domain is defined on ORSLA.

All capability systems enforce the restrictions on the use of object references and allow access to an object to be handed out in a controlled manner, but capability systems have always had a great deal of difficulty revoking access once it has been given out. I adapt the technique described by Redell [Redell74] to ORSLA in section 7.4. It turns out that garbage collection can be a very important tool for revocation. In addition the lists of inter-area links and cables can be used for revocation. In section 7.4 I show how revocation on ORSLA can be as good as the revocation on Multics: a system with excellent revocation abilities.

Finally, the allocation of address space on ORSLA is considered in section 7.5. Capability systems do not allow the user to write programs that allocate address space because such programs can easily violate the low level restrictions on the use of object references. On ORSLA, however, subsystem programmers must be able to write programs that allocate address space so that they can control which objects are adjacent to each other in the address space. Allocation of address space is better known as the allocation of storage in the field of storage management, which has developed a large variety of techniques for allocating storage. Section 7.5 discusses the "allocation of storage" even though the term "allocation of address space" might be more accurate. Section 7.5 describes

three free storage data types that are provided by ORSLA to give programmers a great deal of flexibility in the allocation of storage while preventing anyone from allocating storage that is not free.

7.1 Enforcing the Size of Objects

The obvious way to check each load and store instruction to ensure that the size of an object is not being violated is to have a *size* field in the object reference that specifies the size of the object. The check itself is then very fast. The goal on ORSLA of achieving a small object reference does not allow the use of a large *size* field, however. A small *size* field, on the other hand, will only be able to specify one of a small set of sizes. The size of objects need not be restricted to this small set of sizes, however. If the offset within the object of the word being accessed is less than the size specified in the object reference, then the load or store operation is valid. If the offset violates the *size* field, however, it is not necessary to immediately signal an error. It might be possible for other exact sources of size information to exist that could be checked before an error is signalled. If the exact size of the object is not violated, then the load or store may complete without error.

Thus the *size* field in the object reference will sometimes cover only part of the object. Violating the *size* field in the object reference has the cost associated with it of obtaining the exact size information. The coding of the *size* field in the object reference should minimize this cost. The coding I suggest to achieve a 5 bit *size* field is shown in

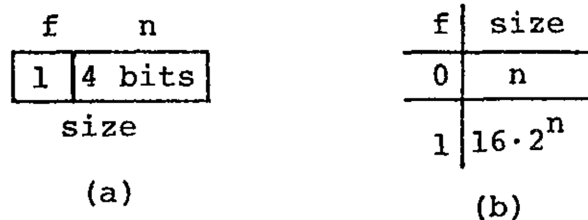


Fig. 33. Coding of a 5 Bit Size Field

Figure 33. The 5 bit *size* field is divided into two subfields: a one bit flag, f, and a four bit integer n, from 0-15. The flag indicates whether the object is large or small. If the object is small, its size is n. If the object is large, its size is $16*2^n$ or 2^{4+n}. This scheme allows objects of size 16 or less to have their exact size in the object reference. An object containing 35 words would have a *size* field that indicates a size of 32 words. Accesses to more than half the words of a large object are approved by this *size* field as long as the object contains less than 2^{20} (about 1,000,000) words.

This coding is quite grainy and it may be argued that being able to cover only a little more than half of an object in some cases is not good enough. By using a slightly larger *size* field and by using the floating point technique shown above, it is possible to reduce the graininess of the coding significantly. For example, the coding for a 7 bit *size* field is shown in Figure 34. The first bit of the field is a flag, f, that specifies how the rest of the field is interpreted. If f is 0, then the rest of the field forms the integer, n, from 0-63, that is the size of the object. If f is 1, then the rest of the field is divided in two: a 2 bit abbreviated mantissa, m, and a 4 bit exponent, e. If both m and e are considered to be integers, from 0-3 and from 0-15 respectively, then the size of the object is given by

---

f	n
1	6 bits

f	m	e
1	2	4

f	size
0	n
1	$16 \cdot 2^e \cdot (4+m)$

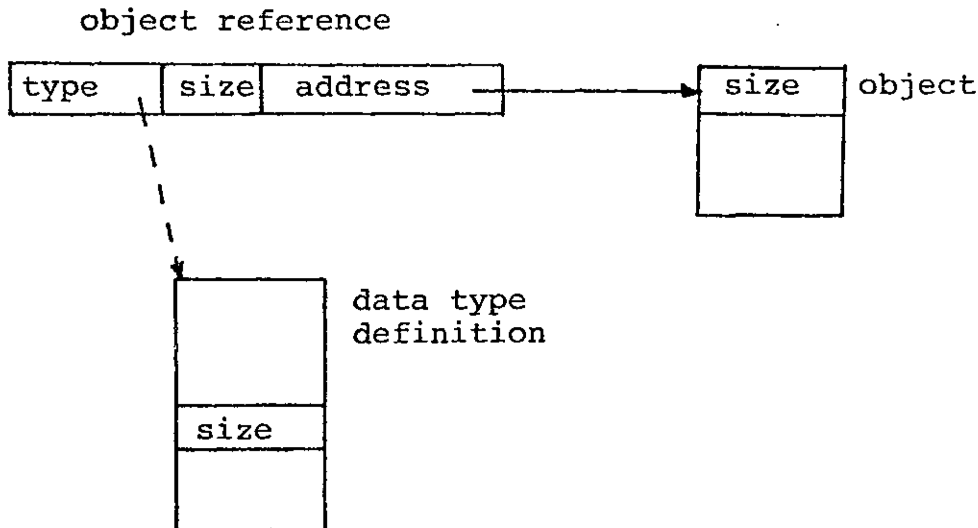
Fig. 34. Coding of a 7 Bit Size Field

$16 \times 2^e \times (4+m)$. A better way to think of it, however, is that m is the low order 2 bits of a 3 bit binary mantissa for a floating point number. The first bit has been omitted because it is always one. If the binary point is to the right of the 3 bit mantissa, then the binary exponent is $4+e$. Thus the smallest number using this floating point format is 64 and the largest is 3,670,016. This *size* field handles the exact size of objects up to 64 words. Objects larger than 64 words are guaranteed to have at least 80% of their size covered by this *size* field. On the average, however, 92% of the size of an object larger than 64 words is covered by this *size* field. This technique of coding the *size* field can easily be used to increase the number of bits in the field in order to achieve larger size ranges and less graininess, but the number of bits in the *size* field cannot be reduced below 5 bits without unacceptable costs. The increase in efficiency that can be obtained by making the *size* field larger than about 9 bits is probably not worth the cost due to a larger object reference. Thus the *size* field on ORSLA contains from 5 to 9 bits.

The small *size* field in the object reference is dependent upon the existence of exact size information elsewhere. An obvious location for exact size information is in the first word of the representation of the object. This location is attractive because it is easily accessible to the CPU and it is not needed unless the object is larger than 16 words, so there is less than a 6% storage overhead for this field. Another possible location for exact size information is in the data type definition. The hardware on ORSLA must be able to map the type code in the object reference into a reference to the data type definition of the object. If all of the objects of a certain type are the same size, then this size can be stored once in the data type definition rather than being stored in each object of that type (see Figure 35).

7.1.1 Protecting System Information within the Representation of an Object

We must remember, however, that the purpose of the *size* field is to enforce the proper use of object references. Once an object has been created, its size cannot be increased by merely modifying the *size* field because the adjacent address space may be used by other objects. There is no point in decreasing the *size* field because all the address space of the original object is still accessible from all the other references to the object.



The three possible locations for size information. Every low level object reference contains a *size* field but the other two fields for exact size information are optional. Their presence or absence is determined by the data type definition.

Fig. 35. Three Locations for Size Information

Thus the size of an object cannot be changed except by the garbage collector, which makes a new copy of the object. Thus the *size* field must remain constant. Mechanisms were described in Section 3.3 for preventing improper modifications to the object reference, so these mechanisms can easily protect the *size* field in the object reference. If size information is stored elsewhere, however, it must be protected from modification also. The most effective mechanism on ORSLA for preventing such modification is storage monitoring. Size information could be stored in the first word of the representation of an object as an atomic object reference whose *storage_monitor* bit is on and whose definition prevents writes. The same technique could be used within the data type definition. Thus the programmer would have access to this information, but would be unable to change it. Furthermore, the garbage collector would not copy this size information explicitly, rather it would be placed into the new copy of the object at the instant the object is created.

Size information is not the only sensitive information that must be stored within the representation of objects, however. There is also the reference count for objects whose reference count is being maintained, and there is the pointer to the data type definition for *escape* data types (see sections 3.5 and 3.4 respectively). If the protection of all this information is designed as a unit, then a cleaner, more efficient design can be achieved.

For example, it is possible for the size and reference count information to be placed into the same word, forming a *size-ref* field within the representation of an object. ORSLA is designed to work with an upper bound of 10^{12}-10^{15} bits of storage, which corresponds to about 2^{34}-2^{44} words of storage. There is no need to allow objects to be larger than 2^{34}-2^{44} words so only 34-44 bits are needed to hold the exact size of an object. A reference count, on the other hand, does not need many bits. Reference counts should only be used when the overhead for maintaining the reference count is not too high. If the count becomes very large, however, then much time is spent incrementing the count to this value and decrementing it to zero. An arbitrarily small reference count field can be used without violating restrictions on the use of object references if, once the count has reached its highest value, it cannot be decremented. I prefer a 5 bit reference count field. A 10 bit reference count field should satisfy the most avid proponent of reference counts, however. If the reference count is combined with the size information in an object reference of type *size-ref* that does not have an address but uses the *size* field and the address field for both the exact size and reference count information, then there will be 11-15 bits available for the reference count depending upon the size of the *size* field in the object reference on ORSLA. The type code used for the *size-ref* field will identify its purpose and will allow the system to maintain the reference count without allowing the user to modify any portion of the *size-ref* field.

The reference to the data type definition contained within an object of an *escape* data type must use an entire word of its own, forming a *data_type_def* field within the representation of the object. Where should this *data_type_def* field be placed, however? For simplicity, let us assume for the moment that there is no *size-ref* field in the object. If the *data_type_def* field is placed in the first word of the representation of the object, then all of the code that operates on the representation of objects of a particular *escape* data

type will avoid use of the first word. If the programmer had decided to make a normal data type, however, then all of the representation dependent code would be different, since it would not avoid use of the first word. This difference in the representation dependent code for *escape* data types and normal data types conflicts with an important use for *escape* data types: debugging a data type that will eventually be a normal data type.

Unfortunately, very few modifications can be made to a data type definition. A bug in a data type definition must often be repaired by defining a new data type in which the bug has been fixed. The old data type then falls into disuse. If the old data type was an *escape* data type, this is inexpensive, while if it was a normal data type, such debugging is very costly because it wastes the limited supply of type codes. Thus *escape* data types should be used when debugging a data type. Finally, when the user is sure that the data type will be used heavily and is also sure that it will no longer be necessary to modify the definition of the data type, then one of the valuable data types in the type code of the object reference can be allocated to this data type. If any portion of an *escape* data type definition needs to be modified in order to enable it to be used as a normal data type definition, however, then more debugging will be needed after conversion to a normal data type. Thus the *data_type_def* field should be invisible to the code that manipulates the representations of objects of the data type. This can be achieved by placing the

---

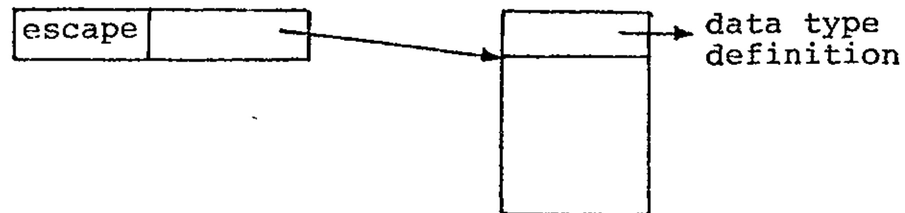


Fig. 36. Escape Data Type

data_type_def field at location -1 in the representation (see Figure 36).

If the *data_type_def* field should be invisible to representation dependent code, what about the *size-ref* field? Representation dependent code may very well want to know what the size of the object is or what its reference count is but this can easily be provided by special *size* and *ref_count* instructions. Reference counts are maintained automatically by the system and so do not require any instructions in representation dependent code to increment or decrement the count. If the existence of the *size-ref* field is invisible to representation dependent code, then reference counting could be requested at the time an object is created even if it causes a *size-ref* field to be added to the representation of the object. Similarly, if the size of one object of a variable size data type happens to be exactly one of the sizes that can be specified by the *size* field in the object reference, it would be possible to omit the *size-ref* field from the object.

To make the existence of the *size-ref* field this dynamic, it is not sufficient to place it at offset -1 within the object, however. It must also be possible to determine the existence of the *size-ref* field directly from the bits of the object reference. This is also necessary to speed up checking the size in the *size-ref* field when the *size* field in the object reference is violated. Furthermore, in order to make the automatic maintenance of reference counts efficient, it is necessary to be able to tell from the object reference whether the reference count is being maintained. This information is kept in the *data_type_info* field in the object reference which was introduced in section 3.4. In addition, the *data_type_info* field must specify whether there is an address in the object reference and also contains the *storage_monitor* bit that specifies whether this object reference is an active storage monitor. The *data_type_info* field therefore needs at least three bits: one for the *storage_monitor* bit and two for an *info* field that specifies one of four mutually exclusive states:

1) there is no address in the object reference
2) there is an address in the object reference but there is no *size-ref* field in the object
3) there is a *size-ref* field in the object but reference counts are not being maintained
4) there is a *size-ref* field and reference counts are being maintained

A system designer may identify more information that must be kept in the *data_type_info*

field, but the size of this field will probably not exceed 5 bits.

The only remaining problem is where to put the *size-ref* field for *escape* data types. Since the *size-ref* field is used more often than the data type definition, I favor a constant offset of -1 for the *size-ref* field while the *data_type_def* field can be placed at -2 when there is a *size-ref* field as well. Thus the *size-ref* field appears in objects as shown in Figure 37.

Now that we have seen exactly how the *size* field is handled on ORSLA, we can see how each load and store instruction will be validated if the *size* field in the object reference is violated. First, the CPU will determine by looking at the object reference whether there is a *size-ref* field in the representation of the object. If so, it will be checked. If there is no *size-ref* field in the object, then the size information in the data type definition is used to perform the final check. Since the size information in both the *size-ref*

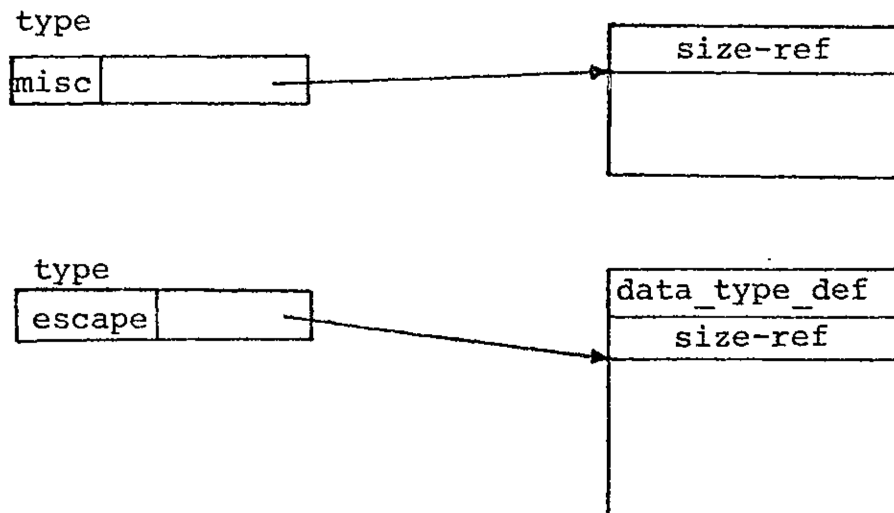


Fig. 37. Location of Size-ref Field

field and the data type definition is exact, violations of either of these fields causes an error to be signalled.

It may have occurred to the reader that this check never allows the programmer to use negative offsets within the object, so the programmer has no way to modify the *size-ref* field or the *data_type_def* field. Thus it may not be necessary to further protect these fields with storage monitors. It cannot be denied, however, that storage monitors do provide further protection. This question really turns on our attitude toward dangling references. If we assume that there are *never* any dangling references, then the use of storage monitors to protect these two fields is unnecessary. ORSLA has been carefully designed so that software cannot generate any dangling references. It is possible for the hardware to generate dangling references, however. For example, an undetected memory error can change a bit in the address of an object reference. This kind of error can be disastrous for a process on an SAV or LAV system since it may lead the program to begin interpreting character strings and machine code as addresses. Such an error is much more limited in effect on ORSLA, however. Since object references are tagged on ORSLA, it is never possible to mistakenly use a character string as an address, or even to use an object reference without full knowledge of the proper representation of the object referenced. Thus the creation of one dangling reference by the hardware does not cause the software on ORSLA to generate more dangling references. If the *size-ref* field or the *data_type_def* field is not protected by storage monitors, then it would be possible for a hardware generated dangling reference to be used to modify these fields, thus allowing all references to the modified object to access storage outside of the representation of the object and possibly modify *size-ref* fields in other objects. Dangling references are like a cancer that, on a single address space system, could easily spread throughout the entire system. The similarity of the effects of dangling references to cancer is what makes elimination of dangling references a significant system reliability issue rather than just a protection issue. ORSLA first prevents the creation of dangling references and then uses context independent object references and *size-ref* fields protected by storage monitors to de-fuse the cancer-like quality of dangling references. Thus even if the hardware does create a dangling reference on ORSLA, the damage caused by it will be very limited. It can be reduced even further by adding a mechanism to ORSLA such that if an expected *size-ref*

field or *data_type_def* field are not where they should be, then the erroneous reference is converted to a reference to the *deleted* object. Thus ORSLA is robust, even though it uses object references extensively.

7.2 Abstract Objects

Most of this thesis has been concerned with enforcing the low level restrictions on the use of object references because enforcing these restrictions is significantly more difficult in a reusable address space than with unique IDs. The low level restrictions ensure that an object will not be accessed without a reference to the object, but they do not allow the operations on an object to be limited: this is the role of the high level restrictions. The basis for the high level restrictions is that an object in the computer is supposed to model an ideal object in a programmer's mind. The ideal object, however, has a strict set of operations defined on it that perform meaningful operations on the object. No other operations may be performed on the object. When a data type is defined a complete set of operations should be defined on it that allow all of the necessary manipulations of the object to be performed in terms that are meaningful to the abstraction of the object. Not all of the users of an object, however, should be able to invoke all of the operations that have been defined on objects of that type. Thus the abstraction given to a user should allow only the operations that are allowed to this user. Enforcing the high level restrictions on the use of object references means sometimes restricting the operations that may be performed on an object even further than the complete set of operations defined on the objects of that type.

For example, a company may want to maintain a data base for inventory control. This data base is a model of the physical inventory the company actually has in its warehouses. The inventory control system automatically places orders for new items when their supply runs low. To keep the model accurate, several people must provide input about changes to inventory and the results of any manual inventories of the stock that are performed. The data base can also be used, however, by auditors, accountants, and executives to find out about the current state of the business. The complete set of operations defined on the data base will support all of these uses. The abstraction of the

data base presented to those who maintain it will include operations for changing or updating the information in the data base. The ability of these people to inspect the entire data base may be limited, however. The abstraction of the data base presented to management, on the other hand, will allow the complete data base to be inspected, but will not allow changes to be made. The ability to present these different abstractions for the same object is entirely consistent with the concept of objects.

It should also be noted that this ability to present different abstractions for the same object and then to enforce the limitations of each abstraction allows the programmer to supply different abstractions of an object, B, to different entities so that only the operations actually used by each entity are defined on the abstraction given to that entity. The entities need never know that they are being treated so carefully until they attempt an operation that they are not supposed to perform.

Access to an object is limited by distributing high level object references for the object and then controlling the operation of switching the *high-low* bit from high to low. This operation must be performed before the representation can be accessed or modified. There have been two methods developed for controlling the conversion of an object reference from high to low: the invocation method and the explicit conversion method. Both of these methods are based on the principle that only the data type definition should be able to perform representation dependent operations on the object. The invocation method, used in such programming languages as Simula 67, Planner-73 [Hewitt73], PLASMA [Hewitt76], and Smalltalk [Kay68], defines an operation f on an object x with the additional arguments $y_1, \ldots, y_n$ to be a procedure call to the data type definition of x. The arguments given to the data type definition are the name of the operation f, the low level abstraction of object x, and the objects $y_1, \ldots, y_n$. The system automatically converts the object reference to x to a low level object reference since it is being passed to its own data type definition. The system is also responsible for finding the data type definition for x given only the object reference to x. ORSLA must therefore be able to convert the type code in the object reference to x into the address of the data type definition of x. In some systems, a data type definition is a complex data structure that contains many different procedures, one for each operation that is defined on the data type. Note that this is

similar to the concept of a *cluster* in CLU [Liskov77], but a cluster is only a syntactic entity in CLU; it does not have a corresponding runtime structure. If the data type definition contains only a single procedure that is given the operation to be performed as an argument, then the programmer has much more flexibility in defining operations on the object and in choosing the precise data structure that must be searched to find the definition of this particluar operation. The invocation method is particularly valuable for defining generic operations that operate on a large number of data types because the operation can be extended to new data types merely by defining the operation in the new data type definitions. The only difficulty with the invocation method is that the data type definition must be searched for the definition of the appropriate operation, which can sometimes take longer than the operation itself, especially for very simple operations.

7.2.1 The Lower Operation

The explicit conversion method makes use of the *lower* operation which converts a high level object reference to a low level object reference. In order to limit the use of the *lower* operation, however, it is given an additional argument: the data type of the object to be lowered. Any program that has an object reference to a data type object that allows objects of that type to be lowered is considered to be part of the data type definition. The explicit conversion method is used much more commonly than the invocation method; it is used by HYDRA and also by the languages Simula 67 and CLU. The origins of the explicit conversion method, however, are lost in the mists of the birth of computer engineering. The *lower* operation is very similar to a simple data type check operation. Data types have been checked whenever multiple representations have been used in computation. The *lower* operation can be used to check data types if it is also a predicate which is *false* if the object is not of the required type. If the check succeeds and the predicate is *true*, the *lower* operation must also return the low level reference for the object. The *lower* operation can both be a predicate and return a value if it sets the condition codes in the machine. The purpose of checking the data type of an object is so that a program will be able to perform representation dependent operations on the object that will have the intended effect. Some programming languages have allowed programs to perform representation dependent operations on an object without even knowing what the

representation of the object is. The *lower* operation merely forces a program to check the data type before performing representation dependent operations. The set of programs that can perform representation dependent operations on objects of a certain type can be limited by limiting the distribution of object references to the data type.

The explicit conversion method requires each operation to search for the data type, while the invocation method requires the data type to search for the operation. The explicit conversion method is thus superior for an operation if the number of operations that can be performed on the object is greater than the number of data types this operation can operate on. The invocation method is superior for an object if the number of operations that can be performed on the object is smaller that the number of data types that those operations are defined on. The *collect* operation used by the garbage collector should be defined with the invocation method since it is defined on every data type on the system. If both methods are supported, it is possible for an operation to check explicitly for the two or three most frequently used and most efficient data types and then use the invocation method for all other data types. Using both methods in this way allows the system to achieve the high speed of the explicit conversion method and the flexibility of the invocation method. Conceptually, both methods are equivalent in that they allow only the data type definition of an object to convert a high level reference to the object to a low level reference. This thesis is concerned with providing hardware support for objects, but is not concerned with the representation of a data type definition, so this thesis will not consider the invocation method further but will consider the explicit conversion method in more detail.

Before further consideration of the explicit conversion method, however, we must consider the access control field in the object reference. As we saw earlier in this section, a single object will have several different high level abstractions. The access control field in the object reference identifies which of these possible high level abstractions is being used by a high level object reference. It was estimated in section 3.4 that between 4 and 10 bits are needed for the access control field. It was also noted in section 3.4 that the access control field is only needed in high level object references, while the *size* field is only needed in low level object references. Since both these fields are about the same size, it is

possible to use the same set of bits in the object reference for both fields. If this is done, however, then the *lower* operation must load the *size* field in the object reference it is lowering and must return the information in the access control field of the high level object reference as an additional value of the operation.

The *lower* operation has now become more than a simple type check but we would like it to continue to be as fast as possible. The *lower* operation takes two arguments: a high level object that is to be lowered and a data type object. The speed of the *lower* operation depends upon what information is kept in the object reference to the data type object. If this object reference contains the type code of the data type, no memory reference will be needed to actually check the type code in the object reference being lowered. How can the time needed to load the *size* field be minimized, however? If objects of a particular type are always the same size, then the object reference to the data type definition could contain a *constant_size* field of 5-9 bits that is loaded into the *size* field of the object reference that is being lowered. If objects of a particular type are of a variable size, however, then the *size-ref* field in the object itself must be accessed to find the size information. If the access control field in the high level reference being lowered is not needed for access control information then it could contain the size of the object. There is no room in the object reference to the data type object for a *constant_size* field or for the type code of the type being defined unless the object reference to the data type object does not contain an address. The hardware on ORSLA is able, however, to convert the type code for the data type into the address of the data type definition. Thus the object reference to the data type definition does not need to contain an address.

A normal data type, i.e. a data type that is not an *escape* data type, is defined by an object of type *normal_type_def*. An object reference to an object of type *normal_type_def* does not contain an address and so never uses an inter-area link. Reference counts cannot be maintained on a data type object, either. An object reference of type *normal_type_def* uses 9 - 16 bits to hold the type code "*normal_type_def*", 3 - 5 bits for a *data_type_info* field, one bit for a *high-low* bit, 5 - 9 bits for an access control field, 9 - 16 bits for the type code of the data type it defines, 5 - 9 bits for a *constant_size* field that may be loaded into the *size* field of a low level object reference of this type when it is lowered, and 2 bits for a

set_size field that specifies where the size information for a low level reference can be obtained. The *set_size* field has three possibilities:

1) size information for the low level reference is contained in the *size* field of the high level reference
2) size information is in the *constant_size* field
3) size information is in the *size-ref* field of the object that is pointed to by the low level reference.

The object reference to an *escape* data type definition is different, however, for two reasons. First, the object reference to an *escape* data type definition must contain the address of the data type definition. Second, the *lower* operation must access the *data_type_def* field in the object whose type is being checked. If the size information to be loaded into the low level reference is in the *data_type_def* field, then no additional information is needed in the object reference to the data type definition. Thus the *data_type_def* field will contain a high level object reference of type *data_type_def* whose *storage_monitor* bit is on to prevent the field from being modified. The access control field contains the size information that is in the *size* field of low level object references to the object. Since the size information in the *data_type_def* field is stored within the object itself, it may reflect the size of that particular object without implying anything about the size of other objects of that type. The address field of the object reference in the *data_type_def* field points to the data type definition. The object reference for an *escape* data type definition contains the type code "*escape*", a *data_type_info* field, *high-low* bit, access control field, and an address field that points to the data type definition. The *lower* operation operates on *escape* data types by accessing the *data_type_def* field in the object being lowered. The address in the *data_type_def* field is compared with the address in the reference to the data type object that is the second argument to the *lower* operation. If the high level reference is lowered, then the *size* field of the low level object reference is loaded from the *size* field in the *data_type_def* field.

7.2.2 Elevate Operations

The *lower* operation converts a high level object reference to a low level object

reference. It must also be possible to convert a low level object reference to a high level object reference since this is the only way the access control field can be set. In addition, an object is first created along with a low level reference to the object. Converting a low level object reference to a high level object reference may be performed by any program without compromising protection because only highly privileged programs can obtain a low level object reference. Actually, it is necessary to have two different operations that perform this conversion:

B = *elevate* (A)

B = *elevate_set_access_control* (A, S)

The *elevate* operation only changes the *high-low* bit in the object reference to A to obtain the object reference to B. Thus the access control field in the object reference to B contains size information about B. The *elevate_set_access_control* operation uses the bit string S for the access control field in the reference to B. The *elevate_set_access_control* operation must be certain that when the reference to B is finally lowered, the access control field will not be used as size information. The current format of the object reference does not keep this information in the reference to A, so the *elevate_set_access_control* operation must access the data type definition of A to make sure it is alright to set the access control field. This memory access could be eliminated if a small change were made to the *data_type_info* field in the object reference. The *data_type_info* field currently contains a two bit *info* field that holds four alternatives. The extra memory access can be eliminated from the *elevate_set_access_control* operation by adding one alternative to these four. This requires an extra bit in the *info* field, but leaves three unused states in that field. The five alternatives in the *info* field would be:

1) no address in object reference
2) no *size-ref* field in object but size information is in access control field
3) no *size-ref* field in object and no size information is in access control field
4) *size-ref* field in object but not maintaining reference counts
5) *size-ref* field in object and maintaining reference counts

Using this coding, the access control field would never be used for size information when the object contained a *size-ref* field. A major mechanism that is not covered in this thesis is reference counting. Since one state in the *info* field is already used for reference counting

even though reference counting has not been described fully, it may be possible that more states in the *info* field will be needed for reference counting. This three bit *info* field has room for four states dealing with reference counting: the equivalent of two bits. It is not clear, however, that this extra bit in the object reference is worth it if it speeds up only the *elevate_set_access_control* operation. This operation is not performed as often as the other operations that are supported by the *data_type_info* field and is only used along with the expense of moderately sophisticated access control. Simple access control needs only the *high-low* bit and may therefore use the *elevate* operation. If the *elevate_set_access_control* operation can be speeded up without using more bits in the object reference, however, then it is clearly worth it.

7.2.3 Access Control Field

Thus we have seen the mechanisms on ORSLA that allow high level objects to be created on which no representation dependent operations are defined. Most of the fields in the object reference serve the needs of the system and the needs of low level objects, but the access control field serves the needs of high level objects. Exactly what information is kept in the access control field and how the field is interpreted is determined by the data type definition. There is no need for the information kept in the access control field to be limited to access control information. The access control field could be considered to be an extension of the representation of the object. Since the system has no control over what the access control field is used for, it is very difficult to decide how many bits should be used for this field. The approach I have taken is to estimate the minimum number of bits necessary by identifying valuable uses for the access control field and then finding representations for those uses that require as few bits as possible.

Throughout this thesis I have frequently placed a piece of information into an object reference in order to avoid a memory reference to the object itself. Although this criterion was sufficient for placing information into the object reference, this criterion is not important when deciding whether to place information into the access control field. The most important reason for this sudden change of design philosophy is that the rest of the object reference is supported by hardware while the access control field is supported by

software. A memory reference is expensive for a hardware supported operation, but is much less important for a software supported operation. Thus information that can be placed within the object itself should not be placed in the access control field.

We have seen that one object can be viewed with different abstractions. If the abstraction that is being used from a particular object reference can be specified in the access control field, then multiple abstractions of the same object can be implemented efficiently. If the access control field did not exist, it would be necessary to create an indirect object for each high level abstraction except the first, which would be the high level definition of the object itself. The indirect object would contain a low level reference to the object and the information that would have been in the access control field if it had been available. Thus use of the access control field saves a great deal of indirection and also saves a significant amount of storage. If the size of the object reference is increased to hold an access control field, however, then the access control field will use a significant amount of storage on the system. Whether the storage saved by use of the access control field would exceed the storage used by an enlarged object reference is dependent upon how ORSLA is used and so would vary from subsystem to subsystem and installation to installation. If the access control field does not enlarge the object reference, however, then the larger the access control field, the more storage is saved. Thus I suggest that the size of the access control field be exactly the same size as the *size* field and that the same bits in the object reference be used for the *size* field and the access control field. We must ask whether this size of access control field is large enough, however.

Regardless of how many bits there are in the access control field, there will always be applications that could take advantage of a larger access control field. The real question is whether enough applications can live with the proposed size of the access control field. If an application discovers that it would like a larger access control field, it may be possible to use a more efficient coding in the access control field that will allow the field that has been provided to be used. The number of bits needed in the access control field can be minimized by using each state of the access control field in references to a particular object to identify an abstraction that is actually being used for this particular object. For example, consider the high level abstractions of a directory object. The abstraction given

to the owner of the directory is the *owner* abstraction, while the abstraction given to an unidentified user is the *unidentified_user* abstraction. The *unidentified_user* abstraction is the abstraction of the directory object that is obtained from the file system when no attempt is made to obtain greater access. The *unidentified_user* abstraction should at least allow users to find out what objects in the directory are being made available to the public and should allow a program to be run that will give access to these objects once appropriate negotiations (possibly between programs) have been concluded. In addition to these abstractions, there will probably be a *sys_daemon* abstraction (a concept from Multics) that allows system utility programs to operate in the directory. There will probably be a *friendly_user* abstraction that allows most or all of the directory to be inspected and allows selected objects within the directory to be used. There might even be a *trusted_user* abstraction that is almost as powerful as the *owner* abstraction. In special situations, it may be necessary to create a special abstraction for a particular person, resulting in the *Sam* abstraction. I have assumed that all of these abstractions are different, i.e. that the sets of operations permitted to the abstractions are different. The name I have given to each abstraction, however, identifies the entities that will use the abstraction. Entities that can make use of the same abstraction are grouped into a class and the name given to the abstraction describes all the entities in the class.

Somewhere within the object or its data type definition the information of which operations are permitted on each abstraction must be represented. This information could be stored in the form of a table within the representation of the object or its data type definition. Each element of the table could be a bit string that defines an abstraction by using each bit to specify the legality of a single operation. It would then be very easy to change the behavior of the abstractions defined by the table. Different abstractions are used by different users, so the operations allowed to a particular class of users can be changed by changing the abstraction they use. This is such a powerful feature that creating two abstractions for two classes of users that happen to have the same behavior is acceptable if the chances are reasonably good that these two classes of users will be given abstractions with different behaviors at some time. Thus, we make good use of the access control field by using it to identify the class of users that will use the object reference and by limiting the number of such classes to those classes that are actually using the object.

The table described above that allows the access control field to be coded efficiently behaves differently from and is much more efficient to use than the data structure that appears on other systems (such as Multics) and is known as the "access control list". The table described above is quickly indexed by the access control field, while an access control list must use information not provided by the reference to an object (usually a user or domain identifier is also needed) and must painstakingly search the access control list to find what abstraction should be presented to the user of the object. Thus although it is necessary for a system to have mechanisms that reduce the frequency of searches of an access control list, the table described above can be consulted once for each high level operation that is performed on the object.

When the very efficient coding for the access control field described above is used, there will probably be very few data types that will need an access control field larger than four bits. An object that cannot use the access control field to distinguish all the different abstractions of the object that are being used will have to use indirect objects. If the access control field is at least four bits, however, the storage overhead for the indirect objects is much less than without an access control field. Assuming a four bit access control field, the access control field in the reference to the indirect object can be used to distinguish 16 different abstractions to the object. Each indirect object could contain the table that defines what operations are used by the 16 abstractions defined by the indirect object, but only one indirect reference would be needed within the indirect object. Thus the amount of storage used by indirect references is now less than the amount of storage needed for the table that defines the abstractions, so the overhead for indirect objects will not be very high. The access control field will actually contain between 5 and 9 bits, however, thus further reducing the number of indirect objects needed by an object that has a large number of abstractions defined on it.

Another example of the use of access control information is provided by areas. The system is able to return a reference to an area given a reference to any object in the area. This can be used either to determine whether two objects are in the same area or to create a new object in the same area as an existing object. The first of these operations may be acceptable regardless of the types of objects involved, while the second operation is more

sensitive. It may be acceptable, however, for the definition of the data type of an object to create another object in the same area as the original object. ORSLA supports these two uses by providing one of two high level abstractions of an area to a program that obtains a reference to the area by giving the system a reference to an object that resides within the area: the *high_level_container* abstraction if the reference to the object in the area was a high level object reference, and the *low_level_container* abstraction if it was a low level object reference. The *low_level_container* abstraction of an area can only be obtained by a data type definition for an object in the area. Each area may define for itself what operations on the area are allowed by these two abstractions.

Another use for the access control field is to hold size information for an object that does not have a *size-ref* field and whose data type consists of different size objects. This use speeds up the hardware supported *lower* and *elevate* operations and has already been discussed. Occasionally, it will be necessary for the representation of an object to place some bits in the object reference. One example of this occurs for machine code objects that have multiple entry points. The address within the object reference must point to the beginning of the machine code object that contains all the code and the references to the constants needed by the code. The object reference for each entry point into the code, however, must indicate in some way which entry point is to be used. The number in the access control field could be used to access a table within the machine code that specifies the bit offset of each entry point within the machine code object.

All of the uses of the access control field to support the high level abstractions of the object can be viewed as merely identifying different abstractions of the object. Many objects will have only one high level abstraction, however. Many more objects will have a few different possible abstractions. These objects may use any coding of the access control field that seems appropriate. Objects that serve as entry points into data bases that contain sensitive or valuable information of some kind may have a large number of different abstractions defined on them. If the data base is used by many people, then there may be a need for many classes of users that may have different abstractions at one time but happen to use the same abstraction at another time. These objects will be forced to use an efficient coding of the access control field and may even be forced to use indirect objects

to increase the effective size of the access control field. There may be a cost associated with handling such large numbers of abstractions, but this cost is acceptable because the cost could not be reduced significantly by enlarging the access control field.

7.2.4 Comparison of Protection on ORSLA with Other Capability Systems

The mechanisms provided on ORSLA to restrict the operations that can be performed on an object are quite similar to the mechanisms on HYDRA. The *high-low* bit on ORSLA corresponds roughly to all the *Kernel_rights* bits on HYDRA, while the access control field on ORSLA corresponds to the *auxiliary_rights* bits on HYDRA. HYDRA encourages the user to encode the legality of a single operation or set of operations in a single *auxiliary_rights* bit, but, as on ORSLA, there is no requirement that any particular encoding be used. The dichotomy between the invocation method and the explicit conversion method that exists on ORSLA is not present on HYDRA. HYDRA at first appears to use a combination of the invocation method and the explicit conversion method since each procedure has an argument template that is matched against the arguments the procedure is called with and "amplifies the rights" of the arguments if the data types of the arguments match the data types in the template. If there is no match then an error is signalled. This is actually a case of the explicit conversion method because a) it amplifies the rights of several capabilities at once (which cannot be done by the invocation method) and b) it does not transfer to a procedure within the data type definition of a capability being amplified. Since an error is signalled if the type codes of the arguments do not match the template, this mechanism is not used to search for the representation dependent code that is needed to perform a representation independent operation. Thus HYDRA does not seem to recognize that the mechanism for amplifying rights could also be used to support multiple representations of the same abstraction.

Another approach to implementing the conversion from a high level object reference to an object reference for its representation has been used in some programming languages [Liskov74] and some capability systems [Redell74, p.80]. Instead of having a *high-low* bit in the object reference, the *lower* operation replaces the type code field of the object reference. This implementation is based on the idea, which has been formalized by Morris

[Morris73], that a user defined data type is based on another, more primitive, object which is its representation. When a high level object reference is lowered, then the result is an object reference for the object that is the representation of the high level object. This representation object, however, is an abstract object itself. If the type of the representation object is a user defined type, then this object has its representation object in turn. Thus there is a chain of representation objects until finally a primitive data type is reached whose operations are implemented directly by the system. If this idea is implemented by simply changing the type code in the object reference whenever an object is lowered one level, then the rest of the object reference uses the representation of the primitive data type. All of the objects that are implemented by hardware on ORSLA, such as integers and floating point numbers, would be implemented directly by such a system, but in addition the system would provide several data types that are general purpose data structures, such as lists, vectors, and arrays. Although this technique isolates the programmer from the physical representation on the machine, the necessity of following the long chains of representations before a primitive data type is encountered is inefficient. Actually, the long chains of representations can be short-circuited by a compiler. Since each data type in the chain has a single representation type, the *lower* operation could be given a data type object that would cause the entire chain to be immediately short circuited to the primitive type. This corresponds to lowering the *high-low* bit on ORSLA. Thus ORSLA encourages a compiled data type definition to contain all of the information in the data type definitions of the representations of the original data type so that the compiled data type definition will manipulate the bits of the low level representation directly. It is likely to be finally determined in the future that a chain of representations is a good way to define data types in high level languages. Machine independence can be obtained by defining an abstraction that allows the fields of the low level representation to be obtained without specifying those fields in terms of bit offsets or sizes. Higher level abstractions can then be defined with this machine independent representation abstraction of the object. A higher level abstraction whose representation is this machine independent representation would execute much more quickly if, when it is compiled, it converts operations on the machine independent representation abstraction into the corresponding operation on the machine's low level representation. If a high level abstraction is built on another high level abstraction, however, then it may be better for the higher level definition, when compiled,

to continue to use the high level abstraction that is its representation since operations on this high level representation may be quite complicated.

A chain of representations as described above is supported on ORSLA by using the access control field rather than the type code field. A good use for the access control field is specifying which high level abstraction of the object is being used. A reference for one high level abstraction is converted to a reference for another high level abstraction rather than being immediately converted to the low level abstraction. Such conversion operations can easily be provided by the data type definition of the object using the mechanisms that have already been described. I am not ready to propose additional mechanisms that will make this way of operating even easier because I believe there is need for additional research before it will be clear what mechanisms should be provided by the system.

The purpose of the chain of representations is to provide representation independence. Consider the operation f, which operates on objects of a certain abstraction, a_1, by lowering that abstraction to a representation abstraction a_2 and performing operations on this lower level abstraction. If a_2 is a high level abstraction, then there may be many different low level representations that could make use of this single definition of f without f being aware of the low level representation. Thus f should not have to determine which low level representation is being used, it should merely determine that a_2 is a reasonable lower level abstraction for the object. The mechanism of changing the type code to convert from a_1 to a_2 does not allow a data type for a_1 to use different representations for different objects of the type. Exactly how this should be implemented, however, is a matter for future research.

Thus the mechanisms on ORSLA support a chain of representations. There is another, more subtle, difficulty with implementing the *lower* operation by modifying the type code. In particular, the references to the representation do not contain the information of what the representation is being used for. The garbage collector can make good use of such information and, in some cases, must have it. For example, a high level abstraction for object w may need access to the objects x, y, and z, but may almost never make use of z. On ORSLA, the garbage collector would be able to find the data type definition for w

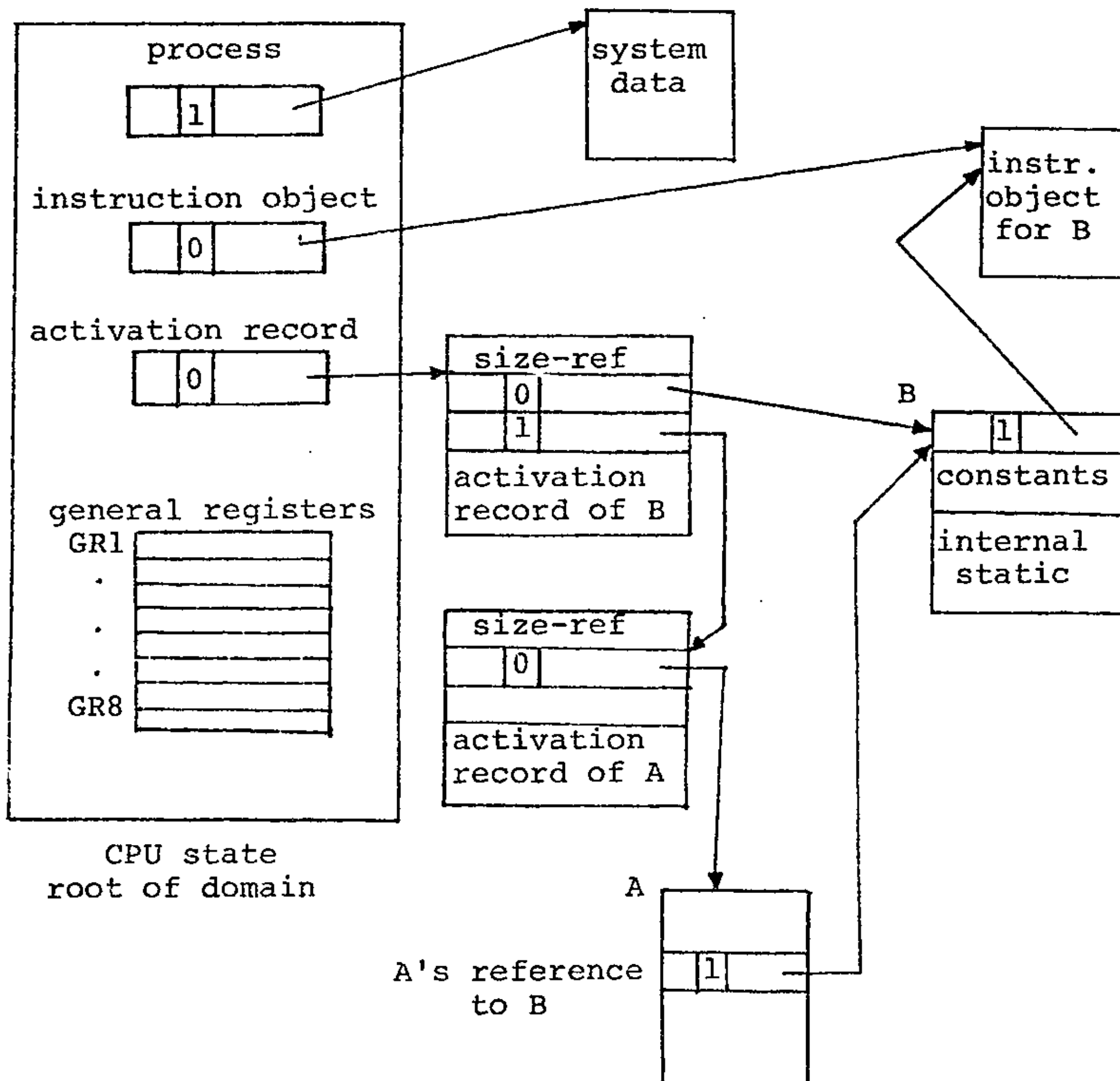
with both high level and low level references to w. The data type definition could specify, in the definition of the *collect* operation, that the garbage collector should place z far away from w, thus increasing locality of reference over the standard copying garbage collector. If we lower the high level abstraction for w by changing its type code, however, it would be natural to represent w with a *vector* of three elements. When the type code in the reference to w specifies *vector*, however, there is no information available about the relative frequency of use of the three elements of the *vector*, so the garbage collector could not do better than the standard copying garbage collector. A more serious case arises with a hash table data type in which the hash index is calculated from the address within the reference to the name of an entry in the hash table [Bobrow75]. If the object that is the name of an entry is moved, the entry must be rehashed into the table. If a vector is used to represent the hash table, and if the first reference to the hash table discovered by the garbage collector is for this vector rather than for the high level hash table, then the garbage collector will neglect to rehash the entries in the hash table thus making it unusable after the garbage collection. These difficulties are very serious, especially when we consider that the garbage collector begins marking the temporary storage of a process from the currently executing activation record. It is thus likely that if a reference to the representation of an object exists, it will be encountered by the garbage collector before a high level reference to the object. These difficulties can be handled, even if we modify the type code when lowering an object reference, by providing primitive data types that are garbage collected properly. If the construction of chains of representations is the only mechanism for type extension, however, then it is not possible to extend the set of primitive data types. To show that such a scheme is practical, it is necessary to show that it will never be desirable to create a high level abstraction that needs, in some sense, a primitive object that was not originally supplied. The approach taken on ORSLA, however, corresponds to providing a mechanism for extending the set of primitive data types. The load and store operations are defined by the hardware for all primitive objects (low level objects) on ORSLA, but other operations, such as the *collect* operation used by the garbage collector, are defined by the data type definition for low level objects (primitive objects) as well as for high level objects. Once the load and store operations have been provided by the hardware, the other operations on a primitive object can be defined in terms of the load and store operations.

7.3 Domains

It is important for a computer system to be able to manipulate all of the objects in memory, however it is not necessary for a process to be able to manipulate all of these objects at every instant. A procedure should only be manipulating its arguments, its activation record, objects such as the root of the file system that are accessible to all programs, its internal static storage, and objects referenced from the procedure. The most valuable feature of protection is that it prevents erroneous programs from doing much damage. Protection should not interfere with legitimate computation, however. An erroneous program can be viewed as an arbitrary sequence of machine instructions. The domain of a process characterizes the amount of damage that could be done by an arbitrary sequence of instructions. The domain of a process at any instant is the set of storage locations that could be accessed and the operations that could be performed on protected objects by an arbitrary sequence of machine instructions executed at that instant. Protection will not interfere with legitimate computation if the domain of a correct procedure includes the objects that the procedure will want to manipulate. Most systems that provide protection have a data structure that specifies the domain. What is this structure on ORSLA?

Unlike most systems, machine instructions on ORSLA do not contain any addresses. They only contain offsets within objects. In order to actually access storage these offsets must be combined with a low level object reference. There are two dedicated CPU registers that contain low level object references: the instruction object register and the activation record register. The instruction counter is an offset within the instruction object. Together they specify the bit location at which the next instruction begins. The instruction object register is not generally accessible to the program. Transfer, call and return instructions allow the instruction object register to be reloaded and the load instruction allows constants to be retrieved from the instruction object, but nothing may be stored into the instruction object. The activation record register points to the procedure's temporary storage. In addition there are several general purpose registers. Conceptually, these registers are part of the activation record. Machine instructions may load and store the general registers and may use any low level object references in general registers to access the representations of

those objects. Since the activation record register contains a low level object reference to the activation record, registers can easily be loaded from and stored into the activation record. Many of the general registers and locations within the activation record correspond to local variables in the procedure. There are two locations that do not and so need more explanation. One location in the activation record contains a low level object reference for



The setting of the *high-low* bit is shown in many of the object references in this figure.
Fig. 38. A Domain

the procedure object. This may be different from the reference to the instruction object. The procedure object will contain the internal static storage for this instance of the procedure and an object reference to the instruction object. The constants for the procedure may be either in the procedure object or in the instruction object depending upon the amount of flexibility that is needed. Another location in the activation record is needed for the return point of the procedure. Usually, when procedure A calls procedure B, it is desirable for B to be ignorant of who A is. This means that B should not be accessing A's activation record. Therefore the object reference in B's activation record pointing to A's activation record is a high level reference. The only operation allowed on this object is *return*. Since activation records are implemented in hardware, the return operation is not slowed down by the fact that it is using a high level reference.

In addition to these sources of information, the processor will contain an object reference to a *process* object which will be accessible to the executing program. This object will allow the program to get a high level reference for the root of the file system and references for other necessary information available to every program.

This completes the sources of information that are available to a program. The domain of a process at a given instant is determined by all the object references that can be reached from any one of these sources of information. The domain of a process is made smaller when any low level object reference in the domain of the program is made into a high level object reference that does not allow access to all of the object references stored in that object.

Are we sure, however, that the domain on ORSLA does not interfere with legitimate computation? Many programming languages, such as FORTRAN, PL/I, and Algol 68, pass arguments "by reference". When a local variable, x, of procedure A is passed by reference to procedure B, it is usually implemented by causing A to pass a pointer to the location in A's activation record that contains the object reference to the value of x. Procedure B may then set the value of x by storing an object reference into A's activation record. There are two ways of implementing this method of call by reference on ORSLA without giving B a low level object reference to A's activation record. Both implementations replace the

pointer to the variable by an object reference to an abstract *variable* object (this object is called a *cell* by some authors). The first implementation uses a high level object reference to the activation record whose access control field specifies that it is abstractly a *variable* object and identifies which location in the activation record is involved. The second implementation requires that a separate *variable* object actually exist outside of the activation record. A reference to this object is given to procedure B. Note that when an array is passed by reference in FORTRAN, PL/I, or Algol 68, a pointer to the first word of the array is passed. These languages do not pass a pointer to the location in A's activation record that contains the object reference to the array. Thus an array is passed by reference merely by passing an object reference for the array. Call by reference is usually used for arrays, but when it is used for other variables, the main purpose is to obtain several return values from the called procedure. Multiple return values can be implemented more efficiently on ORSLA than the mechanisms for call by reference described above. Some implementations of FORTRAN currently implement call by reference of simple variables by using an underlying mechanism that returns multiple values. In the example above, procedure A would pass the object reference that is the current value of x to procedure B which would then return a single value to A which would assign x to this value. Call by reference is the only common programming language feature that might require a procedure to have a low level object reference to its caller. By using the above techniques, we can implement call by reference without causing a procedure to need a low level object reference to its caller, so the domain on ORSLA does not interfere with legitimate computation. Forcing a procedure to use a high level object reference to its caller and high level object references to other procedures called by this procedure drastically reduces the size of the domain of the procedure and allows the domain to be extremely dynamic. The domain on ORSLA does a good job of including only the objects that a procedure really needs and so appears to approach the minimum size that will not interfere with legitimate computation.

7.4 Revocation

On all capability systems the possession of an object reference (capability) by a program allows that program to perform certain operations on the object. Once an entity has distributed a reference to an object, however, it is difficult for the entity to revoke the access granted by that reference. Often, object references passed as parameters to a subroutine are only intended to be of use during the subroutine call. If the called program stores the object references into a permanent data base, however, there is little the caller can do. This same problem occurs if an object reference is mistakenly given to someone who should not have it or if relationships between programmers change so a programmer who was once trusted is no longer trusted. The only way to solve these problems is to introduce mechanisms that allow access to be revoked. ORSLA provides several such mechanisms.

On CAL-TSS and HYDRA it is possible to copy the representation of an object and then destroy the original. This essentially revokes all of the outstanding references to the original object. Although these references still exist, no storage is being used for the original object and no information can be gained from these dangling references. This kind of revocation can be achieved on ORLSA by copying the object and then performing a hard delete on the original object. This causes the next garbage collection to replace all references to the original object with references to the *deleted* object.

7.4.1 Revocation a la Redell

This is not very satisfactory, however, because *all* access to the object is revoked. It would be better if only the access granted by improper references were revoked. A more sensitive revocation mechanism has been developed by Dave Redell [Redell74]. The details of his mechanism are incorporated into the ID map on a CUID system: a mechanism that does not exist on ORSLA. Abstractly, however, Redell's mechanism is an example of indirection. If program A wants to give to program B a reference to object x, and if program A wants to revoke the reference to x when B returns, an indirect object can be used (see Figure 39). Program A would create an indirect object, y, that would contain the reference to x. Program A retains the low level reference to y and passes to B a high level

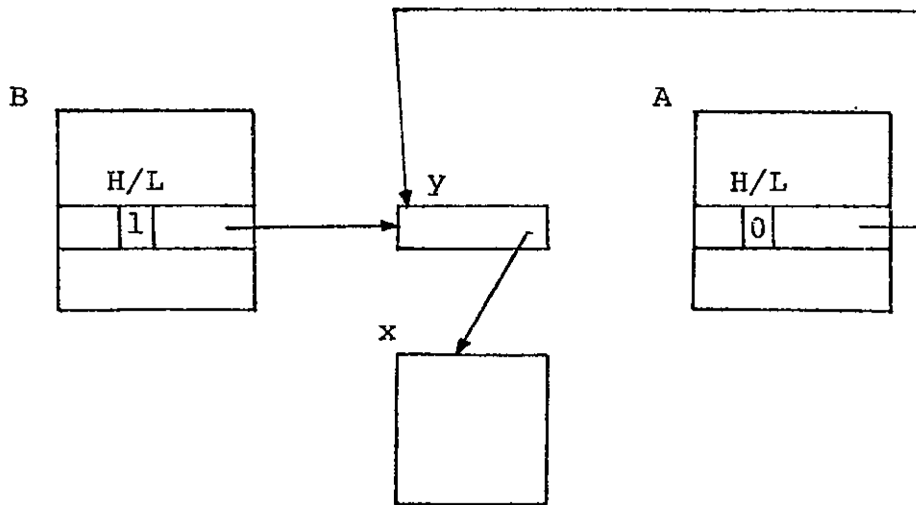


Fig. 39. Revocation via Indirection

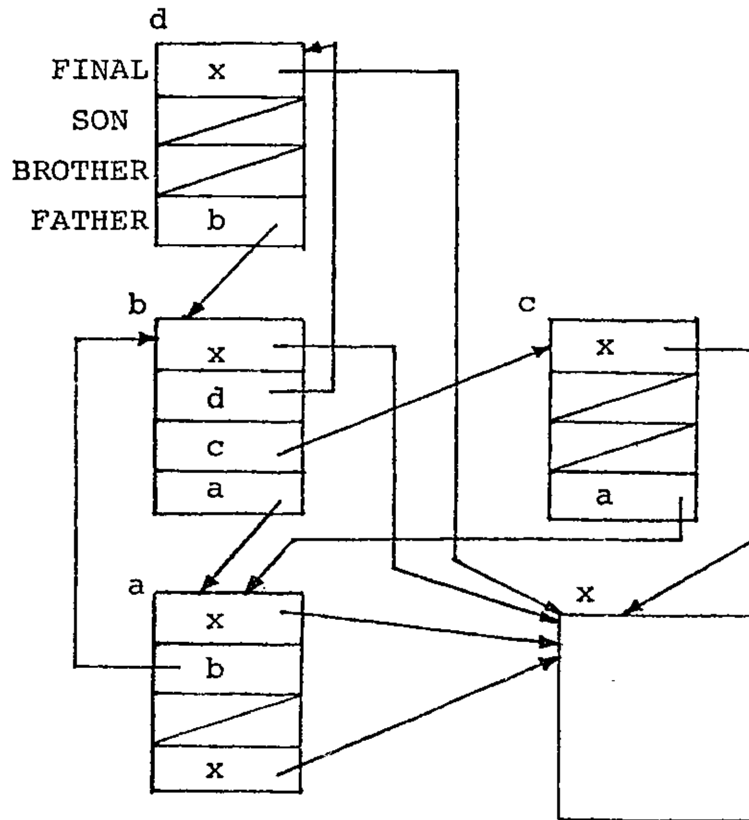
reference to *y*. The high level abstraction of *y* is identical to the abstraction of *x*. Any operations performed on *y* are implemented by performing the same operation on *x*. This can be quite efficient if the indirect object is implemented by hardware. When B returns, then A stores a reference to the *deleted* object in *y* thereby revoking B's access to *x*. The garbage collector could easily convert references to indirect objects containing the *deleted* object into references directly to the *deleted* object. It is not necessary, however, for A to completely revoke all access to *x*. Program A could store a different reference to *x* in the indirect object that would give B more or less access to *x*. This mechanism allows A to completely control the access B has to *x*.

If this mechanism is used heavily, then long chains of indirect objects will be created. Every use of the object would require accessing each intermediate indirect object. This would take a long time and could markedly increase the working set of the computation. Redell's solution is to short-circuit these chains. Much of Redell's algorithm depends upon the use of an associative memory to map object names onto high speed memory. Since this is not done on ORSLA, a more standard method is used here. Redell assumes that the use of an indirect object is more frequent than revocation and so he designs an indirect object that short circuits long chains but correctly revokes all indirect objects. This can be

achieved on ORSLA with an indirect object that contains four object references, named FINAL, SON, BROTHER, and FATHER. FATHER is the reference to the object from which this indirect object was constructed. FATHER may point to another indirect object or, if this indirect object is the last indirect object in a chain, it may point to the object whose access is being controlled. FINAL is the reference to the object whose access is being controlled regardless of how long a chain of indirect objects is. BROTHER points to another indirect object that has the same father as this indirect object. SON points to an indirect object whose FATHER field points to this object. FINAL short-circuits the FATHER chain, while the SON and BROTHER links allow revocation of the short circuits. Revocation is performed by changing the FATHER field. Whenever the FATHER field is modified, all FINAL fields in this indirect object and all its sons, grandsons, etc. are set to their new value. Any SON and BROTHER fields that must also be modified to reflect any new structure in the indirect object chains are also updated.

In Figure 40, program A created an indirect object, *a*, for *x*. This indirect object was passed to program B, which first made an indirect object, *c*, from *a* and passed this to program C. When C returned, B made another indirect object, *b*, from *a* and passed this to program D. Program D made an indirect object, *d*, from *b* and passed it to program E. The indirect objects are now linked together as shown in Figure 40. To revoke an indirect object, *w*, store the *deleted* object in FINAL and then follow the SON link. This reference specifies a tree of indirect objects that are all offspring of *w*. Since *w* appears on the FATHER chain from all these objects, they should be revoked as well. The tree of offspring is found by following both SON and BROTHER links in all the indirect objects found starting with the SON link of *w*. It is also necessary to remove *w* from its list of BROTHERs, after which the *deleted* object can be stored in the FATHER link of *w* as well. If, instead of completely revoking the access to an object, a new reference is stored into the FATHER link of an indirect object, and if this new reference points to another indirect object, *v*, then it is necessary to place *w* on the list of sons of *v*.

Redell's scheme allows restrictions to be placed on the use of an indirect link. This would allow the object reference in the FINAL link of *d* to be different from but less powerful than the FINAL link in *a*. Redell's mechanism depends upon having an access



a, b, c, and d are indirect objects for x.
Fig. 40. Revocation a la Redell

control field in the object reference in which each bit specifies the validity of performing a separate operation. Since this coding is not forced on the user on ORSLA, Redell's mechanism cannot be used. I will leave the development of mechanisms to allow restrictions to be placed on the use of an indirect link to future research.

Indirect objects allow the uses of an object to be broken into classes when references to the object are handed out. These classes form a tree structure. It is possible to revoke access from all members of a particular subtree of users without revoking access from other classes of users.

7.4.2 Revocation Aided by Mechanisms on ORSLA

ORSLA provides another mechanism for dividing users into classes that is more efficient. The access control field in a high level object reference can be used to specify one of a set of from 2^5 (32) to 2^9 (512) different classes of users depending upon the number of bits in the *size* field. It would be possible to maintain a table of bit strings within the object that would specify the allowable operations for each class of user. One bit in each string would specify the validity of one operation. Access for each class of user can then be controlled by setting this table. This method of revocation works when the access control field identifies a class of users rather than a set of operations.

ORSLA can also perform revocation using the garbage collector. Since the garbage collector finds all of the references on the system to a particular object, it would be possible for the garbage collector to selectively substitute references to the *deleted* object for references to an object some of whose access should be revoked. The garbage collector can only be selective if it can distinguish between different references to an object, however. The garbage collector has two kinds of information about a reference. First, there is the object reference itself with its *high-low* bit and access control field. Thus the garbage collector could revoke all low level references to an object, or it could revoke all high level references that identify a specific class of users. The garbage collector is also aware of what area the reference is stored in, however. This is an interesting piece of information for revocation purposes, especially when combined with the access control field and any further division into classes of users caused by indirection. Thus it may be that all low level references for a particular object from areas other than LCAs or the area containing the object should be revoked. Rather than revoking such references, however, the garbage collector could add further levels of indirection a la Redell. This sort of activity by the garbage collector can allow much better use to be made of the 5 - 9 bits in the access control field. When a particular class of users is no longer valid, it can be revoked and its code in the access control field can be reused for another class of users. The garbage collector could also combine several classes of users, possibly making *Sam* access to a certain directory the same as *friendly_user* access thus making the code for *Sam* access available for other classes of users.

These advanced revocation abilities of the garbage collector are no accident. They are due to the fundamental ability of the garbage collector to find all the accessible references on the system to a given object and then to modify all these references. The same mechanism that creates locality of reference and handles fragmentation of storage can also provide sophisticated revocation features. I have tried to describe a few of the ways in which relatively inexpensive revocation can be performed using the garbage collector. The full development of this mechanism will be left to other researchers.

ORSLA provides one last mechanism that can be used for revocation: the inter-area link. It is similar to the indirect object used for revocation a la Redell. Its main purpose, however, is not to provide revocation, so we must accept whatever limitations it may have for revocation. It is possible to revoke inter-area links to an object without garbage collecting the area containing the object. It should be noted that most of the references to a permanent object that will need to be revoked are not stored in the same area as the object. The list of inter-area links immediately identifies the references to an area from other reasonably permanent data bases. Each area is used by the class of programs and users that use the information contained in the area, so some information about the class of users of an inter-area link can be obtained by looking at the area that the link comes from. If this does not correspond to the class of users identified by the access control field of the reference, then a protection violation may have just been discovered after-the-fact. Although this would be distressing, the violation must be due to an error by a user or a program in handing out references to the object, not to a fault in the system. It is better for the system to aid in the detection and correction of such errors than to prevent their discovery.

7.4.3 Access Control Lists on ORSLA

Multics is a protection system that does a good job of revocation. Each segment on Multics has an access control list that specifies what access each user has to the segment. To revoke access, the access control list is merely modified. Since a segment on Multics corresponds to an area on ORSLA, access control lists on Multics could be approximated on ORSLA by providing an access control list for each area that would specify the legality

of all incoming links and cables. Although inter-area links come from other areas rather than from other users, each area is under the control of one or more users. If ORSLA can identify these users, then the access control lists on ORSLA could be similar in appearance to the access control lists on Multics. The access control lists on ORSLA could also specify the access allowed from individual areas as well as the access allowed from areas controlled by a particular user or class of users. These access control lists would be checked whenever an inter-area link or cable is created, thus they are not only a mechanism for revocation. When the access control list for an area A is modified, however, then all incoming links and cables to A are checked against the new access control list and any illegal links are revoked. If any cables to A are found to be illegal, the area from which the cable comes is garbage collected and all the direct references to objects in A are revoked. Thus modifying the access control list on ORSLA can provide revocation that is as sudden and as complete as on Multics.

Unfortunately, it is not easy to tell what users control an area on ORSLA. Some indication of what user controls an area can be gained from what directory controls the area. Another hint of which user controls the area can be obtained from the account that is being charged for the storage used by the area. A third possibility is to create another field in the area object that would hold a highly privileged object reference to the user who is responsible for the area. Presumably users would control the distribution of such object references very tightly, so the system could use this field to determine what user controls an area and at least have the assurance that the only other users who could control the area would be users that are trusted by the user who is responsible for the area. Exactly how ORSLA would determine what user controls an area will be left to future research, but whatever method is used will probably not identify what user controls an area much better than the techniques mentioned above.

Although the access control lists could control what LCAs are allowed to have cables to the area, it should be remembered that the existence of a cable from an LCA does not imply uncontrolled access by the LCA. It may be that the user who owns the LCA may have access only to high level object references to the objects in our area. When an operation is performed with a high level object reference, the owner of the LCA loses

control of the process while the operation is being performed. Thus even the fact that a low level reference to the object exists in a hostile LCA is no indication that protection has been compromised. If the rings on Multics are ignored, then the access control lists on Multics can be seen as preventing processes from using segments. This corresponds on ORSLA to controlling cables from local computation areas. Multics, however, allows a process to have limited access to a segment while ORSLA provides this via access control lists by controlling high level references from permanent areas.

Access control lists on ORSLA form a separate protection mechanism that uses the concepts for specifying protection that have been developed on Multics, i.e. people having access to areas. There are two reasons why access control lists form a secondary protection mechanism that will support the primary mechanism provided by object references. First, there are uncertainties in identifying which people have access to a given area. These uncertainties exist even on Multics, although the uncertainties may be greater on ORSLA. Second, the protection provided by object references is extremely fine-grained and there is a surprisingly small amount of difficulty in specifying and enforcing the variegated control it provides. Access control lists on ORSLA can, however, provide secondary support to the protection provided by object references. A major drawback of protection by object references is that the flow of information must be pre-planned. When a program gives out a reference to an object, the program must have some idea of what class of users will use the reference so the class of users can be identified in the object reference. If the program does not distinguish the various references to the object that it distributes, then everyone will always have the same amount of access to the object. Access control lists, however, are very dynamic, allowing the user to decide upon access control issues and to identify classes of users after references to the objects have been distributed. Furthermore, the concept of controlling peoples' use of areas is easy for people to understand. The concept of distributing access to people via object references is also easy to understand, but it is not easy to appreciate the implications of the further distribution of object references. Access control lists, however, do not allow further distribution of access; this inability is a strength when access is being distributed manually, although it becomes a weakness when more careful thought is given to the control of access. Finally, the fact that the criteria for determining legality of a reference is completely different for protection by object

references and protection by access control lists means that protection by access control lists can be used as a backup method for catching the failures of protection by object references even when the control of access has been thoroughly planned in advance. By dealing with areas rather than individual objects, it is possible for access control lists to detect unprotected references to any of the objects in a data base, while the largest effort in protection by object references will be concentrated on the objects that are entry points into the data base. Thus protection by access control lists can be set to catch any blatant lapses in protection that occur with the carefully designed protection by object references. In this case, the access control lists should be set to prevent only those references that the protection by object references would never allow when working properly.

7.4.4 Resnapping Mistakenly Revoked Object References

A fundamental problem of revocation is that it is difficult to identify precisely the references that should be revoked. Although this thesis proposes mechanisms for separating references into classes, there will probably be several references in each class, only some of which should be revoked. The classic answer to this problem of capability systems is that if an object reference (capability) is mistakenly revoked, the user should be able to get another copy of the object reference. This answer, however, ignores the fact that the user may not be aware of the existence of the object reference (capability) that was revoked. In Chapter 6, I presented a mechanism for unsnapping and then resnapping inter-area links. This mechanism could be used to give the user some help in obtaining new copies of a reference that has been mistakenly revoked. At the very least it would identify what object had been previously referenced and give some clue as to where to get another copy of the reference to the object. Unfortunately, this mechanism for resnapping links has serious protection implications that have not been discussed here. Future research should present and work out the protection problems of dynamic linking on capability systems.

The existence of areas and of garbage collection present many opportunities for sophisticated revocation. One of the reasons revocation has been such a problem on capability systems has been the lack of garbage collection. Once all the references to an

object can be found and modified, sophisticated revocation is possible. Exactly how it should be carried out and what controls need to be placed on it are subjects that have been touched lightly here, but which need more research. Much of this work can only be done after more of the details of ORSLA have been worked out.

7.5 Allocation of Storage

The allocation of storage has always been a complicated subject [Knuth68]. A large number of techniques have been developed for allocating storage. Unfortunately, allocating address space is a very sensitive job on a capability system. If unique IDs are used, however, one per object, in a non-reusable address space, then allocating them is very simple. HYDRA and CAL-TSS each have a single system-provided function that allocates IDs for objects. In a re-usable address space, however, where locality in the address space is important, no single allocation algorithm would be acceptable in all cases. Therefore it is necessary to allow the subsystem programmer to write storage allocation routines. We must assume, however, that any code a programmer writes may have a bug in it. If a storage allocator were to allocate the same piece of storage to two different objects, it would violate a fundamental assumption of the system: that an object reference really points at the object it is supposed to point at. Some method must be devised that allows subsystem programmers to write storage allocators but prevents them from allocating the same piece of storage twice.

The immediate purpose for allocating storage is to create a new object. To create the object *x*, which is of type *t*, the programmer will call the procedure *construct_t*, which is responsible for creating objects of type *t*. The arguments to *construct_t* specify the initial state of *x*. *Construct_t* is considered to be part of the definition of *t*. *Construct_t* first computes the number of words needed for the new object and then calls a storage allocator which returns a free storage object, *f*, containing the required number of words. *Construct_t* now converts *f* into the low level abstraction of *x* using the *allocate* operation that is defined on free storage objects. The *allocate* operation initializes any *size-ref* field or *data_type_def* field that may be in the object, zeroes the rest of the object, and creates the initial low level object reference for *x*. At the same time, *f* is modified so that this piece

of storage cannot be allocated again without first being freed. *Construct_t* now has a low level reference for x, so it uses this reference to initialize the representation of x and then returns the object that it has finished constructing.

Thus we see that the existence of free storage data types allows the information encoded in *construct_t* to be limited to information about the representation of objects of type t. All of the complexities of storage allocation have been left in the storage allocator. Furthermore, it is not necessary for the storage allocator to contain any information about type t. Can free storage data types, however, allow the storage allocator to operate? Surprisingly enough, there are really only two operations on free storage objects that are needed to implement storage allocation: splitting a free storage object into two or more free storage objects, and combining two adjacent free storage objects into a single object. In addition, however, a storage allocator must be able to use the storage in the free storage objects to maintain the data base that allows the allocator to keep track of the free storage objects. This data base is usually called the "free storage list".

Now that we have seen how free storage objects are used, we are ready to consider how we can prevent the same piece of storage from being allocated to more than one object. We have assumed that all of the objects on ORSLA are using disjoint pieces of address space. If we include free storage objects in our concept of objects, then it would be natural to extend this requirement to free storage objects as well. Thus a word of address space is either free, allocated, or inaccessible. It is allocated if it is part of the representation of an accessible object other than a free storage object. A word of address space is free if it is part of an accessible free storage object. A word of address space is inaccessible if it is not part of an accessible object. The reference count machinery converts an allocated object into a free object at the moment when the object becomes inaccessible. Garbage collection will eventually convert inaccessible words into free words. No word of address space may be part of more than one accessible object. If this requirement is followed, then it is obvious that only storage that is free can be allocated regardless of how many bugs an individual storage allocator may contain. There are three operations on free storage objects which could, if not designed carefully, violate the disjointness of all objects. These operations are: *allocate*, *combine_storage*, and *split_storage*. Each of these operations

return a new object that shares storage with one or more operands of the operation. These operations must, therefore, destroy the old objects before creating the new ones so that at no point will two objects exist that contain the same word of storage. In order to ensure that this restriction is followed, all of the free storage data types are defined by the system. On the other hand, it is the system's responsibility to ensure that enough different representations for free storage objects are provided so that no limitation will be placed on the possible storage allocators by limitations in the representations of free storage objects. I suggest that the following three free storage data types will provide the needed flexibility.

7.5.1 First Free Storage Data Type

The simplest kind of free storage type, FS_1, consists of objects of different sizes whose representations are made entirely of free storage. Since the reference to an FS_1 object points directly to free storage, however, this kind of object is destroyed only when it becomes totally inaccessible. Fortunately, there is no need for references to exist to objects that have been combined or split or allocated. Thus each of the *allocate*, *combine_storage*, or *split_storage* operations will only accept an FS_1 object as an argument if the operation has been given the last accessible reference to the FS_1 object. The operation merely destroys this reference to the FS_1 object in order to make it inaccessible. The operation can only check that it actually has the last remaining reference to the FS_1 object if reference counts are maintained on FS_1 objects. Thus a *size-ref* field is needed in all FS_1 objects.

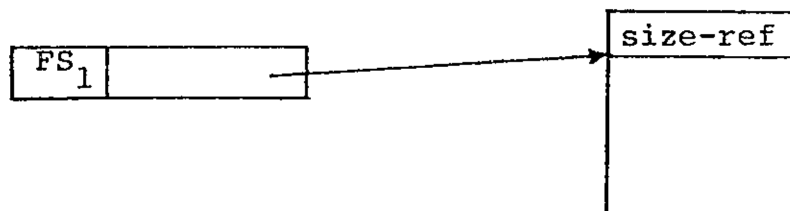


Fig. 41. Simple Free Storage Type (FS_1)

Unfortunately, this thesis does not describe exactly how reference counts are maintained; this has been left for future research. Only the *allocate*, *combine_storage*, and *split_storage* operations are defined on the high level abstraction of an FS_1 object, but the load and store instructions are defined on the low level abstraction, allowing storage allocators to use the free storage in FS_1 objects for maintaining the free storage list.

Storage is freed either by the garbage collector or by the reference count machinery. The old information stored in newly freed storage must be erased before the storage allocator may look at it, so all storage is "zeroed" out when it is freed. Actually, instead of zeroing the storage, it is filled with a special atomic monitor object that allows information to be stored in the monitored location, thus erasing the monitor, but causes the *uninitialized_storage* interrupt if the monitored location is read from. Filling storage with this monitor is called *cleansing* the storage. If a storage allocator uses only one or two words in each free storage object for the free list, then cleansing the storage in a free storage object when it is allocated would be largely redundant since the storage was cleansed when it was freed. To avoid this unnecessary operation, the *size* field in the object reference of a low level FS_1 object only covers the part of the object that is being used by the storage allocator. The *size* field in a low level FS_1 object reference can only be modified, i.e. decreased or increased up to the size of the object, when the reference count is 1, i.e. when the object reference being modified is the only one is existence.

We can now see how the *combine_storage* operation is performed on FS_1 objects:

A = *combine_storage* (B, C)

The *combine_storage* operation is only permitted when the storage for B and C is adjacent. If B precedes C, then the *combine_storage* operation is permitted only when the reference count of C is 1. Then the reference to C is destroyed, the *size-ref* field of B is increased by the size of C (*size-ref* + 1), and the dirty storage in C is cleansed. Note that there may be other references to B when this operation is performed. B is merely enlarged to include C.

There is a requirement on ORSLA that the representation of every object must be entirely contained in one area. This allows an entire area to be freed at once when a garbage collection has finished. To achieve this the *combine_storage* operation must check

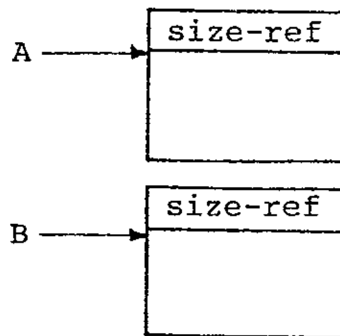


Fig. 42. Combining Free Storage

that both B and C are part of the same area if the point of joining occurs exactly at a page boundary. If the point of joining does not occur exactly at a page boundary then B and C must both be in the same area because the page on which the join occurs is only in one area.

The *split_storage* operation is used to reduce the size of free storage objects.

(A, B) = *split_storage* (C, I)

The FS_I object C is reduced in size until it has only I words in it. Object C is transformed into object A, while B is an FS_I object for the storage removed from C. The *combine_storage* and *split_storage* operations are used by storage allocators to construct pieces of free storage of the proper size for constructing objects.

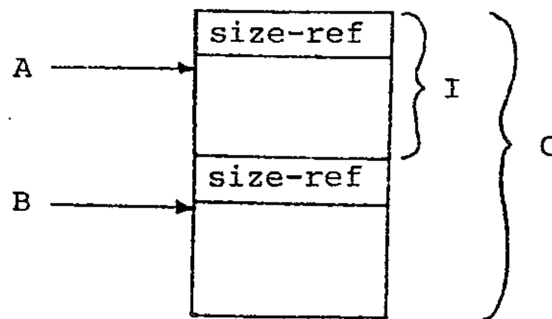


Fig. 43. Splitting Free Storage

Once a free storage object of the proper size has been obtained by a program that is constructing an object, the *allocate* operation is used to convert the free storage object into the type of object being constructed.

X = *allocate* (A, D)

Object A is a free storage object whose reference count is one, i.e. the *allocate* operation is given the last remaining reference to A. Object A is converted to an object of type D. All of the dirty storage in A is cleansed and a low level object reference of type D is constructed to the storage in A. This new object reference is a reference to object X, which is of type D. If X contains a *data_type_def* field or a *size-ref* field, these fields are placed in X by the *allocate* operation.

The FS_1 type of free storage object is useful in many circumstances. Many storage allocators will operate using only the FS_1 type of free storage. Most storage allocators, when invoked to allocate some storage, will return a free storage object of type FS_1. Programs that create objects by using the *allocate* operation will usually be converting a free storage object of type FS_1 to the type of object they are creating.

7.5.2 Allocating Address Space and Storage to Areas

Although the FS_1 data type is very useful, it is necessary for the system to provide a set of representations that will be adequate for all applications. Before we can consider more representations for free storage objects, however, we must take a closer look at an interesting feature of ORSLA; address space and physical storage can be allocated separately. Since the machine language level of ORSLA operates in a virtual address space, however, free storage objects on ORSLA actually control allocation only of address space. Storage is allocated by these objects only when there happens to be storage associated with the address space that is being allocated. There are no physical storage objects on ORSLA that control physical storage that is not assigned to a section of the address space. Any object that contains a *data_type_def* field and/or a *size-ref* field must have storage associated with the words of address space that contain these fields, but it is possible for the rest of the object to consist of words of address space that do not have any storage associated with them.

Ultimately the system is responsible for the control of physical storage. It decides which piece of physical storage to use for a piece of address space and decides when to move the piece of address space to another piece of physical storage. Physical storage is broken into pages, so whenever there is a word of storage associated with a word of address space, there are also words of storage associated with the rest of the words of address space in the page of address space. The idea of allocating storage separately from address space in this way is not new. Multics allocates address space in large, fixed blocks: one segment at a time. A page of storage is allocated to a segment whenever the user stores information into a page of address space that does not have any storage associated with it. Allocating storage and address space separately allow certain large variable size objects to be implemented efficiently. If a large array can change size, such as a stack, then it is convenient to allocate a relatively large amount of address space to the object while only allocating enough pages of storage to accomodate the current size of the object. This technique is particularly valuable for allocating address space. An area may initially obtain a large block of address space to allocate from. The area then allocates this address space to objects starting from one end. Storage need not be associated with the part of this address space that is still free. Whenever blocks of address space are allocated separately, there can be no assurance that they will be adjacent to each other in the address space. Each block of address space, when it has been allocated to objects, will probably contain a small amount of internal fragmentation at the end of the block. This inefficiency is minimized by allocating large blocks of address space to areas.

Although most programmers will allocate free storage objects from areas, storage management within an area is a complicated, application dependent task that the system cannot specify for all applications. Thus the system provides three operations that allocate free storage objects to areas and also allow storage to become associated with allocated sections of address space that do not have storage associated with them.

f = *sys_alloc* (*area*, *nw*)
sys_alloc_address_space_only (*area*, *nw*, *f*)
assign_storage (*area*, *page_number*)

The variable *f* is a free storage object while *area* is the area to which the allocation is being made. Whenever an allocation is made to an area, the quota is checked and the usage information in the area is updated, including the list of pages of address space allocated to the area. The catalog of pages maintained by the system must also contain a reference to the area that each page is part of. Although these operations are given the number of words (*nw*) of storage and/or address space to allocate, the system only allocates pages of storage and/or address space. *Sys_alloc* allocates address space that has storage associated with it, while *sys_alloc_address_space_only* allocates address space that does not have storage associated with it. Both of these operations return a free storage object that reflects the number of words actually allocated, which may be larger than *nw* but will not be less. The *assign_storage* operation is used to assign storage to address space that has already been allocated but does not have storage assigned to it. The *assign_storage* operation checks that the page of address space is part of the area and does not yet have any storage assigned to it.

In addition, there are three operations that allow an area to return storage and address space to the system:

> *sys_free* (*f*)
> *retrieve_storage* (*x*, *offset*, *nw*)
> *hard_delete* (*area*)

The *sys_free* operation frees address space and any associated storage. The free storage objects passed to the *sys_free* operation must match up with page boundaries since the system will only free pages that are entirely contained in the free storage object. The *retrieve_storage* operation does not free address space, but removes no more than *nw* words of storage from the object *x* starting at the *offset* word within *x*. The object *x* does not need to be a free storage object. The *retrieve_storage* operation only retrieves whole pages of storage but will not retrieve the storage associated with a word of address space that contains a *size-ref* field or a *data_type_def* field. The *hard_delete* operation returns all of the address space and storage allocated to the area to the system. When an object is freed by the reference count machinery, a small amount of address space may have been freed,

so the resulting free storage object is given to the area that contains the object. The area may combine it with adjacent free storage objects and may eventually allocate its address space to a new object.

7.5.3 Second Free Storage Data Type

There is another free storage type (FS_2) that is also rather simple but is designed to handle free address space that does not have any storage associated with it. An FS_2 object is two words long. The first word contains the address of the first word of the block of free address space. The second word contains the size of the block of address space. This free storage object does not need a reference count. It does not matter how many references there are to the two word object because the two word object is considered to be allocated storage. Only the storage pointed to from the two word object is considered to be free. The only pointer to this free storage is kept within the two words of the FS_2 object, so there can never be more than this one pointer to the free storage itself, thus there is no need to maintain reference counts to the free storage. The *split_storage* operation on FS_2 objects can take a positive or negative integer. If positive, the address space is allocated from the beginning of the block. If negative, it is allocated from the end of the block. The *split_storage* operation generates a free storage object of type FS_1, however. Another

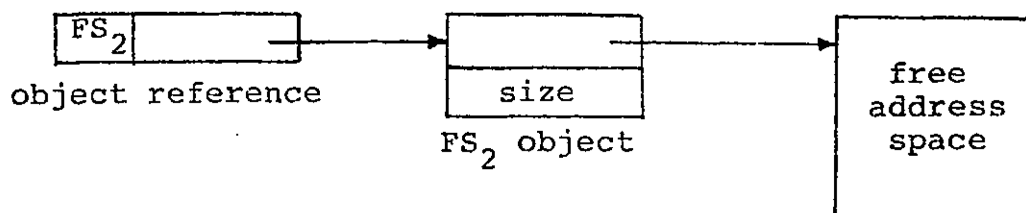


Fig. 44. FS_2 Data Type

operation, *split_storage2*, is needed that will create a new block of type FS_2. FS_2 objects can be reused whenever the size of their free storage block reaches zero, so the *split_storage2* operation reuses an existing FS_2 object rather than creating an additional two word FS_2 object. *Split_storage2* (A, I, B) allocates the storage from A and combines it with B. Neither A nor B can be FS_1 objects since the operation does not return any values. The *combine_storage* operation, if given an FS_1 and an FS_2 object, places all the storage in the FS_2 object. Given two FS_2 objects it places all the storage in its first argument.

Objects of type FS_2 are ideal for allocating storage from a large block of address space that may not have any storage associated with it. FS_1 objects are much better for returning storage from a storage allocator because they contain only free storage. If an FS_2 object is returned from a storage allocator, then the two word FS_2 object cannot be safely reused because the user will have a reference to it.

7.5.4 Third Free Storage Data Type

The FS_1 and FS_2 data types handle many of the needs for storage allocation. There is a third representation, FS_3, that can achieve great efficiency under certain conditions. This is similar to FS_2 in that the representation for the object is not free storage. It has a *size-ref* field. The first two words are the same as the words in an FS_2 object. Unlike FS_2, however, storage is not allocated by changing these two words. Instead, an FS_3 object contains a bit string that specifies which words in the block pointed to from the FS_3 object are actually free and which are not free. The block pointed to from the FS_3 object is called the block of possibly free storage. The number of words represented by each bit are specified by a number in the fourth word of the FS_3 object. There is a requirement that all contiguous blocks of address space be entirely contained within one area. To aid in satisfying this requirement the third word of an FS_3 object is a reference to the area that free words in the block managed by the FS_3 object belong to. The rest of the words of the FS_3 object contain the bits that identify which parts of the block of possibly free storage are free and which parts are not.

When an FS_3 object is created, the bead size, area specification, and the block of possibly free address space must all be specified. The block of possibly free address space

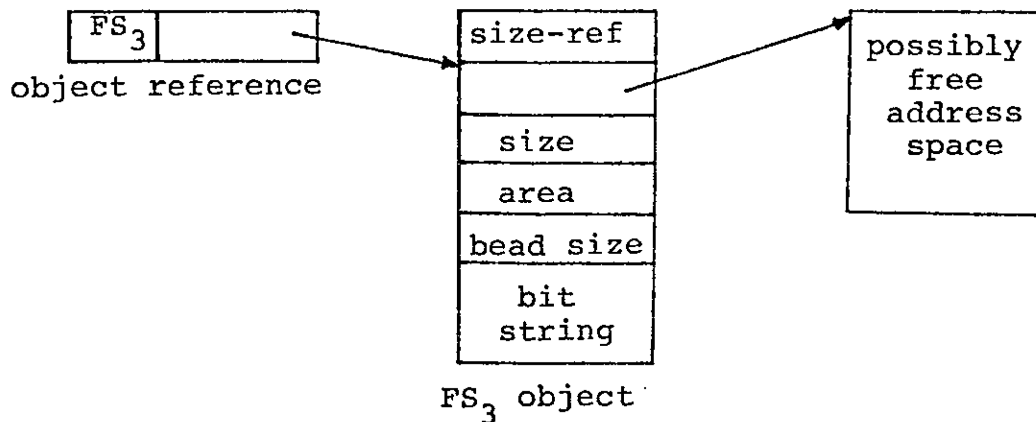


Fig. 45. FS_3 Data Type

can either be specified with integers, in which case the bit string is set to indicate that none of the address space is free, or the block may be specified with an FS_1 or FS_2 object in which case the block of free storage is removed from the FS_1 or FS_2 object and the bit string in the FS_3 object is set to indicate that all of the address space is free. In either case the bead size multiplied by the number of bits in the bit string must be greater than or equal to the number of words in the block of possibly free storage.

The *combine_storage* operation, when given an FS_3 object and either an FS_1 or an FS_2 object, takes storage from the FS_1 or FS_2 object that is included in the block of possibly free storage of the FS_3 object and changes the corresponding bits in the FS_3 object to indicate that this storage is free. The bead size determines a certain graininess in the handling of storage. If a free storage object does not exactly correspond to the bead boundaries then only those bits will be set to "free" for which the corresponding words are *all* known to be free. Sloppiness about bead boundaries results in storage being lost to the allocator; it does not result in overlapping objects. Whenever a free storage object is combined with an FS_3 object such that one extreme of the object corresponds to an exact bead boundary and an exact page boundary then the area the free storage is part of is compared with the area in the FS_3 object.

The *split_storage* operation is not defined on FS_3 objects. Two more operations are

needed instead.

A = *allocate_from_3* (*f*, *offset*, *i*)

The *allocate_from_3* operation returns an FS_1 object. The arguments *offset* and *i* specify bits within the bit string that correspond to the words that are to be allocated. The first bit is specified by *offset* and the number of bits to allocate is *i*. All *i* bits must indicate that these words are free. All these bits are set to "allocated" and a free storage object is returned that corresponds to the bits that have been modified. An operation is also needed that will produce FS_2 or FS_3 objects.

allocate_from_3_to_2 (*f*, *offset*, *i*, *a*)

The *allocate_from_3_to_2* operation is similar to the *allocate_from_3* operation except it combines the allocated storage with *a*, which is either an FS_2 or an FS_3 object.

An FS_3 object is best for handling large amounts of fragmentation, especially when the bead size is larger than one. In this case searching the bit string can be done without causing excessive paging. It is also easy to determine if a bead being freed can be recombined with another bead. The bit string can be read by the user of the FS_3 object, but cannot be modified except by the combine and allocate operations. The FS_3 data type is used by the system to keep track of free pages of address space.

7.5.5 Conclusion

The definitions of all three data types preserve three properties. First, no single block of free storage crosses area boundaries. Second, every piece of free storage is counted exactly once; no two free storage objects overlap. Third, when a free storage object is converted to another data type, no dangling references are created and the object is no longer considered to be free storage. These properties prevent any bugs that may exist in a storage allocator from creating dangling references.

The interface chosen for a storage allocator causes an object constructor to communicate a bare minimum of information to the allocator. If the allocator is malicious or has a bug in it, the worst thing that can happen is that the allocator could never return. If the allocator returns with a free storage object of the correct size, then the worst that can happen is that the free storage object cannot be converted into an object of another type.

Once the conversion is complete, however, the storage allocator can no longer interfere with the computation. The allocator can no longer access the storage that was allocated and the storage has been cleansed for the user thereby preventing information from previous computations from leaking through the storage allocation machinery. The user need not clean the storage to protect himself from the storage allocator, either.

Appendix A
The Size of the Address Space

Now that we are familiar with the entire structure of ORSLA we are ready to consider a problem that arose very early in the thesis. ORSLA uses just one address space for the life of the system, but how large must this address space be? Throughout the thesis I have assumed that in order to handle 10^{12} bits of storage (about 2^{34} words of 59-82 bits each) it is necessary to have only 40 bits in an address (2^{40} words of address space). The purpose of this appendix is to justify this assumption.

The main purpose of the address space is to allow all of the on-line storage to be used quickly and easily. The address space on ORSLA must be designed for a maximum amount of online storage. If this maximum is exceeded it is possible that the system will not be able to make effective use of the additional physical storage. Physical storage is expensive, especially in quantities as large as 10^{12} bits. If the users of a system go to the expense of buying that much storage, the languages and operating system should enable the users to make use of it in any way they see fit. This should include putting all (or most) of the storage into one very large block. 10^{12} bits of storage would require 34 bits just to be able to address all the words of storage. ORSLA allows single blocks of address space as large as 2^{34} words.

2^{34} words of address space are not enough, however. For one thing, the user needs to be able to allocate blocks of address space without assigning storage to the address space. The best use of this technique is for the activation record stack. The contiguous block of address space in the stack increases the efficiency of the use of storage. It is easy for this feature to get out of hand, however. Given a particular pattern of usage, a certain ratio will develop between the total address space allocated (A) and the total storage allocated (S). This is the A/S ratio. The total address space needed to handle 2^{34} words of storage will be $(A/S)*2^{34}$. This ratio increases the size of an address on ORSLA. It is difficult to decide what A/S ratio is needed to use ORSLA effectively, but an A/S ratio as small as 2 may be acceptable. This requires only one extra bit in the address. I will use this A/S ratio as an example, but a different ratio could easily be used.

Regardless of what A/S ratio is chosen, the user will be tempted to use a higher ratio. To prevent this it is necessary to have address space quotas on areas as well as storage quotas. The total address space quota will be equal to twice the total storage quota. Let us see what effect such a small A/S ratio has on the selection of a stack size.

When a stack is created on ORSLA it is necessary to obtain a certain amount of physical storage quota for the use of the stack. The amount of storage quota obtained must be somewhere between the average size and the maximum size of the stack. It is only necessary to obtain the average size if an infinite number of independent stacks are all drawing from the same pool of storage quota. If a stack is not sharing a pool of quota with other data structures, then it is necessary to obtain enough storage quota for the maximum amount of storage the stack will use. Since ORSLA does not set any limit on the maximum size of a stack, the amount of available quota will determine the maximum size of a stack. In practice, a stack will draw from the same pool of storage quota as several other data structures but usually these data structures will not be independent. Often all of the data structures will grow large at the same time. This means that the amount of storage quota that must be obtained just for the stack is larger than the average size of the stack but somewhat less than the maximum size. Again, the available quota may well determine the maximum size. Once it has been determined how much storage quota is being added to the pool of quota in order to handle the stack, the initial block in the stack section should be made twice this size. This will allow the stack to become quite unusually large before it is necessary to create a new section of the stack. If it can be predicted that a data structure will grow, then we can allocate twice as much address space to it as the size to which we expect it to grow because we would have already obtained an amount of storage quota equal to the size to which we expect the data structure to grow. If many data structures do not use more address space than storage, then it may be possible for some data structures to have high A/S ratios even if the average A/S ratio is limited to two.

Fragmentation of the address space is a serious problem in a large, long-term address space. Areas will obtain blocks of address space of various sizes and will later return these blocks. Holes of free address space will develop between the blocks of allocated address space. It will not be possible to actually allocate all of the address space to areas. Some parts

of the address space will be broken into unusably small pieces. Assuming an upper bound of 10^{12} bits on the total amount of online storage and an A/S ratio of two, we want to be able to allocate a total of 2^{35} words of address space. How large must the entire address space be to ensure that we will always be able to allocate 2^{35} words of address space in the size blocks that we desire?

This question was faced by J. M. Robson in 1971 [Robson71]. The answer depends in part upon the range of sizes of the blocks of address space that must be allocated. Robson assumes that the smallest block of address space that can be allocated is one word, but ORSLA never allocates units of address space that are less than a page, so I will deal with units of one page: 2^9 words. We want to be able to allocate a total of 2^{26} pages of address space in blocks as large as all physical memory: 2^{25} pages. Robson uses the notation N*(M,n) to specify the smallest address space necessary to allow a total of M pages to be allocated with a maximum block size of n pages. If n is a power of two ($n = 2^{25}$), then Robson comes to the conclusion that

$$4Ma/13 - 16*2^a/13 + 9M/13 \leq N*(M, 2^a) \leq M(a + 1)$$

Substituting our numbers gives us:

$$4*2^{26}*25/13 - 16*2^{25}/13 + 9*2^{26}/13 \leq N*(2^{26}, 2^{25}) \leq 2^{26}(26)$$

$$1.94 * 2^{28} \leq N*(2^{26}, 2^{25}) \leq 1.63 * 2^{30}$$

In other words, if we choose an address space of 2^{31} pages (2^{40} words) we can guarantee that storage fragmentation will never prevent ORSLA from allocating any size block of address space. The allocation strategy that Robson suggests is a slight variant of the buddy system, so it should be reasonably easy for ORSLA to use an address space allocation strategy that will never get it into trouble. In general, in order to be able to allocate a total of $A = 2^a$ words of address space and $S = 2^s$ words of storage on an ORSLA system with a page size of 2^p words, the size of the address space must be about A(s-p+1) words.

Throughout the thesis, I have made a great effort to reduce the size of the fields in

the object reference. I have made a good case that in order to handle 10^{12} bits of storage, it is necessary to have an address space containing between 2^{35} and 2^{40} words. I have advocated selecting the larger size for several reasons. First, storage fragmentation will require a larger address space than 2^{35} words, but exactly how much more address space will really be needed to handle the storage fragmentation that will actually occur on ORSLA will not be known until an ORSLA system is actually built. Furthermore, the storage allocation algorithm used by ORSLA will affect the amount of storage fragmentation, but the cost of running a storage allocation algorithm that minimizes fragmentation may be prohibitive. The algorithm described by Robson that results in his upper bound on the amount of fragmentation is so similar to the buddy system that I expect this amount of fragmentation to be economical to achieve. Lastly, the uncertainty in the upper bound on the amount of physical memory that ORSLA should have is much larger than the uncertainty in the fragmentation of storage. By being conservative on the amount of storage fragmentation, we build in a small safety factor into the upper bound on all storage. Thus if we design ORSLA so it is rated for 10^{12} bits of storage, it may actually be able to operate with more storage, but not much more storage. Exactly how far ORSLA can be pushed will not be known until it is built and its limit tested.

Appendix B
The Area Object

The area object is one of the few objects on ORSLA that is defined by the system. The lists of inter-area links are very sensitive system-maintained data structures that the system depends upon. When the area object was first presented in Chapter 4, only some of the uses of an area were apparent and so some of the information in an area object could not be described. This appendix provides a complete description of the area object.

An area object resides in the area it defines. In fact, the area object is the first object created in the area; as a result, the area object always begins at a page boundary. Thus the area specifier for each page stored in the page map need not use a full word address; only a page address is needed. The rest of the system refers to an area with a full object reference, as with other objects.

Many of the fields in an area object contain information that must be correct if the system is to operate properly. The owner of an area will often be able to affect these fields, but only in very restricted ways. Other fields within the area object are less sensitive and so may be set by the owner. Although some of the behavior of an area object is defined by the system, the owner of an area is given as much flexibility as possible to define the rest of the behavior of the area. The items in the area object are:

1) list of named objects - This is a list of character string names together with object references to the objects they name. The objects named should reside within the area. All of the objects in the area should be ultimately accessible from the named objects in the area. The automatic mover will move other objects out of the area.
2) controlling directory - This is the directory that controls this area. This item allows the pathname of objects in this area to be generated.
3) free list - This is a list of blocks of free storage and address space. When storage and address space are allocated from this area, the storage in this list decreases. The area may request that ORSLA allocate blocks of address space and storage to the area to increase the amount of storage in the area's free list.
4) address space quota - This is the maximum number of pages of address space that

can be allocated to the area at any one time. The user is able to set the quotas on an area. The user can be sure of being able to use this much address space, but cannot get more without changing the quota.

5) address space used - This is the total number of pages of address space currently allocated to the area by ORSLA. This number may never exceed the quota.

6) pages allocated - This is a list of the pages of address space allocated to the area. When an area is deleted, e.g. after it has been garbage collected and all the accessible data moved into a new copy of the area, all of these pages of address space are returned to ORSLA along with any pages of storage associated with these pages of address space. This list does not enable any of the address space to be accessed nor does it identify any objects that may reside in the address space.

7) storage quota - This is the maximum number of pages of physical storage that may be assigned to pages of address space that have been allocated to the area.

8) storage used - This is the number of pages of physical storage currently being used by the area.

9) incoming links - This is the list of inter-area links from other areas that are referencing objects in the current area. This list is used in garbage collection and is automatically maintained by ORSLA.

10) incoming cables - This is the list of cables from other areas to this area.

11) outgoing links - This is the list of inter-area links in the current area referencing objects in other areas. This list is also used in garbage collection and is automatically maintained by ORSLA.

12) outgoing cables - This is the list of cables from this area to other areas. This is actually a hash table of outgoing cables if there are more than three cables leaving this area.

13) area information - This field contains several bits that specify information about the area. The bits in this field are: the *LCA* bit to specify whether the area is a local computation area, the *garbage_collecting?* bit to specify whether the area is being garbage collected, the *lib* bit to specify whether the area is a LIBRARY area, and some bits to specify whether objects may be moved automatically in or out of the area and if not, whether objects that are accessible from elsewhere in the system but not from the current area should be deleted or retained in the current area.

14) miscellaneous information - This is a list of miscellaneous information that is associated with the area but is not sensitive information. It includes several garbage collector data bases when the area is being garbage collected and contains the list of objects that are to be moved to another area by the manual mover. The user of an area may keep information on this list.

15) access control list - This is a list specifying what object references are allowed from other areas to objects in this area. This list will often identify other areas by the users or classes of users that control that area and will often prohibit large classes of object references, such as all low level object references. This list will also specify whether objects may be moved automatically to or from these areas and, if the automatic mover is interfered with, whether an object that is inaccessible from this area but cannot be moved elsewhere should be deleted or retained in this area.

13) lock - Each area may be manipulated by any of the processes on the system. During manipulation of the area, it may be necessary for changes to be made in the free list or in the lists of incoming or outgoing links in this area. To prevent parallel processes from interfering with each other, this lock must be set whenever these lists are read or written so that only one process at a time will manipulate the lists.

References

[Baecker70] Baecker, H. D. Implementing the Algol 68 Heap. BIT Vol. 10, No. 4 (1970), pp405-414.

[Baecker72] Baecker, H. D. On a missing mode in Algol 68. *SIGPLAN Notices* Vol. 7, No. 12 (Dec. 1972), pp20-30.

[Baecker75] Baecker, H. D. Areas and record-classes. *The Computer Journal* Vol. 18, No. 3 (Aug. 1975), pp223-226.

[Baker77] Baker, Henry G., Jr. List processing in real time on a serial computer. M.I.T. A.I. Lab. Working Paper 139, Cambridge, Mass. (Feb. 1977), 30p.

[Batson77] Batson, A. P. and Brundage, R. E. Segment sizes and lifetimes in Algol 60 programs. *Comm. ACM* Vol. 20, No. 1 (Jan. 1977), pp36-44.

[Berry76] Berry, D. M., Erlich, Z., Lucena, C. J. Correctness of data representations: Pointers in high level languages. *SIGPLAN Notices* Vol. 8, No. 2 (1976), Conference on Data: Abstraction, Definition and Structure, pp 115-119.

[Birtwistle73] Birtwistle, G. M. et al. *SIMULA BEGIN*. Auerbach Publications, 1973, Philadelphia.

[Bobrow72] Bobrow, D. G. et al. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM* Vol. 15, No. 3 (March, 1972), pp135-143.

[Bobrow75] Bobrow, D. G. A note on hash linking. *Comm. ACM* Vol. 18, No. 7 (July 1975), p413.

[Burks46] Burks, Arthur W., Goldstine, Herman H., von Neumann, John. Preliminary discussion of the logical design of an electronic computing instrument. 1946, in *Computer Structures: Readings and Examples* by Bell, C. Gordon and Newell, Alan, McGraw-Hill, 1971, New York, pp92-119.

[Constantine68] Constantine, L. L. Segmentation and design strategies for modular programming. *Modular Programming*, ed. by Tom O. Barnett, Information & Systems Press, Cambridge, Mass., 1968, pp23-42.

[Dahl72] Dahl, O. J., Dijkstra, E. W., Hoare, C.A.R. *Structured Programming*,

Academic Press, New York, 1972.

[Data76] Conference on Data: Abstraction, Definition and Structure. *SIGPLAN Notices* Vol. 8, No. 2 (1976).

[Denning68] Denning, Peter J. The working set model for program behavior. *Comm. ACM* Vol. 11, No. 5 (May 1968), pp323-333.

[Denning70] Denning, Peter J. Virtual memory. *Computing Surveys* Vol. 2, No. 3 (Sept. 1970), pp153-189.

[Dennis65] Dennis, Jack B. Segmentation and the design of multiprogrammed computer systems. *JACM* Vol. 12, No. 4 (Oct. 1965), pp589-602.

[Dennis66] Dennis, Jack B. and Van Horn, Earl C. Programming semantics for multiprogrammed computations. *Comm. ACM* Vol. 9, No. 3 (March, 1966), pp143-155.

[Deutsch76] Deutsch, L. Peter and Bobrow, Daniel G. An efficient, incremental, automatic garbage collector. *Comm. ACM* Vol. 19, No. 9 (Sept. 1976), pp522-526.

[Dijkstra68] Dijkstra, Edsger W. The structure of the "THE" multiprogramming system. *Comm. ACM* Vol. 11, No. 5 (May, 1968), pp341-346.

[Eastlake69] Eastlake, D., et al. ITS 1.5 Reference Manual. MIT AI Lab Memo No. 161A (July 1969). Cambridge, Ma. 175p.

[Fabry71] Fabry, R. S. List-structured addressing. Ph.D. Thesis, Univ. of Chicago, 1971.

[Fabry74] Fabry, R. S. Capability-based addressing. *Comm. ACM* Vol. 17, No. 7, (July, 1974), pp403-412.

[Fenichel69] Fenichel, R. R. and Yochelson, J. C. A LISP garbage collector for virtual-memory computer systems. *Comm. ACM* Vol. 12, No. 11 (Nov. 1969), pp611-612.

[Feustel72] Feustel, E. A. The Rice Research Computer - a tagged architecture. *SJCC*, 1972, pp369-377.

[Feustel73] Feustel, E. A. On the advantages of tagged architecture. *IEEE Computers* C-22, No. 7 (July 1973), pp644-656.

[Fisher70] Fisher, David A. Control structures for programming languages. Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon Univ., May, 1970, 206p.

[Fry76] Fry, James P. and Sibley, Edgar H. Evolution of Data-Base Management Systems. *Computing Surveys* Vol. 8, No. 1 (March, 1976), pp7-42.

[Greenblatt74] Greenblatt, Richard. The LISP machine. M.I.T. A.I. Lab. Working Paper 79, Cambridge, Mass. (Nov. 1974), 13pp.

[Hammer76] Hammer, Michael. Data abstractions for data bases. *SIGPLAN Notices* Vol. 8, No. 2 (1976), Conference on Data: Abstraction, Definition and Structure, pp 58-59.

[Hewitt72] Hewitt, Carl. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. AI TR-258, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, April, 1972, 408p.

[Hewitt73] Hewitt, C., Bishop, P., and Steiger, R. A universal modular Actor formalism for artificial intelligence. *IJCAI-73*, Stanford: Stanford University, August, 1973, pp235-245.

[Hewitt76] Hewitt, Carl. Viewing control structures as patterns of passing messages. A.I. Memo 410, M.I.T. Artificial Intelligence Lab. (Dec. 1976), 57p.

[Hoare73] Hoare, C.A.R. Recursive data structures. Stanford AI Lab Memo AIM-223, Stanford University (Oct. 1973), 32 pp.

[Hoare75] Hoare, C.A.R. Data reliability. *Int. Conf. on Reliable Software*, 1975, in *SIGPLAN Notices* Vol. 10, No. 6 (June 1975), pp528-533.

[Honeywell72] The Multics PL/I Language. Honeywell Information Systems, 1972.

[Jones76] Jones, Anita K. and Liskov, Barbara H. A language extension for controlling access to shared data. *IEEE Trans. on Software Engineering* Vol. SE-2, No. 4 (Dec. 1976), pp277-285.

[Kay68] Kay, Alan C. FLEX, a flexible extendible language. Computer Science Dept. Technical Report 4-7. University of Utah. June, 1968.

[Kieburtz76] Kieburtz, Richard B. Programming without pointer variables.

SIGPLAN Notices Vol. 8, No. 2 1976, *Conference on Data: Abstraction, Definition and Structure* (March 1976), Salt Lake City.

[Kilburn62] Kilburn, T. et al. One-level storage system. *IRE Trans.* EC-11, Vol. 2, (April 1962), pp223-235.

[Knight74] Knight, Tom. CONS. M.I.T. A.I. Lab. Working Paper 80, Cambridge, Mass. (Nov. 1974), 22p.

[Knuth68] Knuth, Donald E. *Fundamental Algorithms*, Vol. 1 of *The Art of Computer Programming*, Addison-Wesley, Reading, Mass. 1968.

[Lampson76] Lampson, Butler W., Sturgis, Howard E. Reflections on an operating system design. *Comm. ACM* Vol. 19, No. 5 (May 1976), pp251-265.

[Licklider65] Licklider, J.C.R. *Libraries of the Future* M.I.T. Press, Cambridge, Mass. 1965.

[Lindsey71] Lindsey, C. H. Making the hardware suit the language. in *Algol 68 Implementation* ed. by J.E.L. Peck, North-Holland Pub. Co., Amsterdam (1971), pp347-365.

[Liskov74] Liskov, B. H. and Zilles, S. Programming with abstract data types. *SIGPLAN Notices* Vol. 9, No. 4 (April 1974), pp50-60.

[Liskov77] Liskov, Barbara, et al. Abstraction mechanisms in CLU. Computation Structures Group Memo 144-1, M.I.T. Dept. of Elect. Eng. and Computer Science, (Jan. 1977), to appear in *Comm. ACM*, 31p.

[Lomet75] Lomet, D. B. Scheme for invalidating references to freed storage. *IBM J. Res. Develop.* (Jan. 1975), pp26-35.

[Madnick74] Madnick, Stuart E. and Donovan, John J. *Operating Systems*, McGraw-Hill, New York, 1974.

[McCarthy65] McCarthy, John et al. *LISP 1.5 Programmer's Manual.* 2nd ed. M.I.T. Press, Cambridge, Mass. 1965.

[Morris73] Morris, James H., Jr. Protection in programming languages. *Comm. ACM* Vol. 16, No. 1 (Jan. 1973), pp15-21.

[Organick72] Organick, Elliot I. *The Multics System: An Examination of its Structure.* MIT Press, 1972, 392p.

[Organick73] Organick, Elliott I. *Computer System Organization - The B5700/B6700 Series.* Academic Press, 1973, New York.

[Palme73] Palme, Jacob. Protected program modules in Simula 67. Research Institute of National Defense, Sweden, July, 1973, NTIS * PB-224 776, 25p.

[Redell74] Redell, David D. Naming and protection in extendible operating systems. MAC-TR-140, Massachusetts Institute of Technology, 1974.

[Ritchie74] Ritchie, Dennis M., Thompson, Ken. The UNIX Time-Sharing System. *Comm. ACM* Vol. 17, No. 7 (July 1974), pp365-375.

[Ross67] Ross, Douglas T. The AED free storage package. *Comm. ACM* Vol. 10, No. 8 (Aug. 1967), pp481-492.

[Schroeder71] Schroeder, Michael D. Performance of the GE-645 associative memory while Multics is in operation. ACM Workshop on System Performance Evaluation, ACM, N.Y. (April 1971), pp227-245.

[Steele75] Steele, Guy L., Jr. Multiprocessing compactifying garbage collection. *Comm. ACM* Vol 18, No. 9 (Sept. 1975), pp495-508.

[Wadler76] Wadler, Philip L. Analysis of an algorithm for real time garbage collection. *Comm. ACM* Vol. 19, No. 9 (Sept. 1976), pp491-500.

[Wegbreit74] Wegbreit, Ben. The treatment of data types in EL1. *Comm. ACM* Vol. 17, No. 5 (May 1974), pp 251-264.

[Wulf74] Wulf, W. et al. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM* Vol. 17, No. 6 (June 1974), pp337-345.

Biographical Note

Peter Bishop was born in York, Pennsylvania on February 9, 1949. He studied Electrical Engineering at the University of Rochester from September, 1966 to January, 1969. While at the University of Rochester he was a system programmer for an IBM 7700 - EAI 680 hybrid computer system.

Mr. Bishop completed his undergraduate work in Electrical Engineering at the Massachusetts Institute of Technology in June, 1970. He then began graduate work in Computer Science at M.I.T. and received the B.S. and S.M. degrees in Electrical Engineering in June, 1972 and the Ph.D degree in June, 1977. While at M.I.T., Mr. Bishop helped found the Student Information Processing Board and developed an accounting system for students using Basic on Multics. Mr. Bishop was also a system programmer on Multics for awhile.

While working on his doctorate, Mr. Bishop spent two years helping to design and to implement the Planner programming language. The first implementation was done on Multics in PL/I and the second implementation was done on ITS in LISP.

Mr. Bishop was a teaching assistant in 1974, first assisting in an elementary course on programming languages, but later taking full responsibility for a graduate course at M.I.T. entitled Programming Language Processors.

Mr. Bishop will now join Xerox in Palo Alto, California.