WORKING PAPER 111

September 1975

Garbage Collection

in a

Very Large Address Space

by

Peter B. Bishop

Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract

The address space is broken into areas that can be garbage collected separately. An area is analogous to a file on current systems. Each process has a local computation area for its stack and temporary storage that is roughly analogous to a job core image. A mechanism is introduced for maintaining lists of inter-area links, the key to separate garbage collection. This mechanism is designed to be placed in hardware and does not create much overhead. It could be used in a practical computer system that uses the same address space for all users for the life of the system. It is necessary for the hardware to implement a reference count scheme that is adequate for handling stack frames. The hardware also facilitates implementation of protection by capabilities without the use of unique codes. This is due to elimination of dangling references. Areas can be deleted without creating dangling references.

This research was done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology and was supported by the Office of Naval Research under contract number N00014-75-C-0522.

Very Large Address Space

Experience with the Multics system indicates that a large virtual address space makes the system easier to program. The virtual address space allows multiprogramming to occur with greater efficiency and also allows a program to be used with a wider range of hardware configurations. A remaining problem to ease of use on Multics is the fact that each process has a different address space and therefore cannot communicate pointers from one process to another. If the system had a single address space for the life of the system that would contain all direct access storage on the system, then pointers could be stored in files or in inter-process communication areas and could easily be used by any process on the system. This would make the system easier to program and would also markedly increase the speed of certain wide classes of computations.

Fabry [6] has discussed the advantages of using absolute addresses in data bases and has compared such a scheme with many other addressing schemes. Fabry suggests that paging should be eliminated in favor of segmentation. This results in the operating system being forced to do jobs that could be performed better by the user and his subsystem, such as keeping track of the size of an object, garbage collection, and increasing locality of reference. The approach taken here is to eliminate segmentation in favor of paging. This results in a very large virtual memory that is surprisingly similar to conventional computers except that the problems of relocation, input/output to disk, and storage overlays have been eliminated. The problems of storage management, and allocation and freeing of storage remain as do their traditional solutions [12]. All the work done on locality of reference is applicable to this kind of virtual memory [18].

Current Techniques

Unfortunately, garbage collection in a very large address space is a very difficult job. At the present time, if a garbage collection is done, it must include the entire address space. If the address space includes all the users and is very large, then all user computations must be suspended while the garbage collection runs. Since the address space is very large, the garbage collection will take a long time. In addition, the garbage collector will probably have a large working set, causing it to thrash and thereby make a big job even bigger. This is unacceptable on any time-sharing system or any other system that has even modest real-time requirements.

Some people have been hoping that a technique of garbage collection could be developed that could be run at the same time as normal computation [22] [4]. Knuth [12] indicates this is Minsky's idea (exercise 2.3.5-12). Then a separate processor could be devoted solely to garbage collection. The problem with this approach is that this separate processor would still be touching memory that is not used in the other executing processor; it would have a large working set which would use up a fair amount of the high speed memory. If the garbage collector thrashed, it would also clog the disk channels. On top of these problems, there would be memory interference with the other processor. Recently such a garbage collection algorithm was proposed [22]. It has a fair amount of overhead for the garbage collector and also for the computation that is running in parallel with the garbage collector.

Overview of Proposed Technique

My approach to the problem is to decrease the amount of work done by the garbage collector. I suggest it is possible to garbage collect a small piece of the address space without looking at the rest of it. This approach has been tried with limited success by H.D. Baecker [1] [2]. Greenblatt [8] uses "areas" and "invisible pointers" and claims to be able to garbage collect these areas separately, but does not describe the technique. A. Rochefeld [20] also used a technique that allowed a small piece of the address space to be garbage collected separately. If the address space is broken into pieces called *areas*, and if each area is used for one logical purpose, such as for a data base, as files are now used on most computer systems, then lists of inter-area links can be maintained. Each area can then be garbage collected separately from all other areas, since the list of links pointing into the area are immediately available without searching the rest of the address space. Each process would have a separate area, called the *local computation area*, for its stack and temporary storage. Greenblatt has a similar area that he calls the working storage area. Areas used for data bases that can be manipulated by many concurrent processes must have storage allocation procedures that set locks before allocating storage. Since a local computation area is only used by one process, it is not necessary for that process to set locks when allocating storage from its local computation area. In PL/I, the term "area" is used to refer to a contiguous block of storage [10]. On many systems the object from which to allocate storage may consist of many separate blocks. This is what is meant by an "area" here. It is similar to the "zone" of the AED free storage package [21] and the "mutual" of LISP 2 [23] in this respect. This is in contrast to the "segment" on Multics which is only one block.

Since areas can be garbage collected separately, processes can also be garbage collected separately by garbage collecting the local computation area for the process. Since the garbage collection involves relatively small pieces of storage, a copying garbage collection can be used. This will allow locality of reference to be increased by a garbage collection [7] [15]. The technique of using an area for one logical purpose increases locality of reference all by itself. Since the garbage collection takes place in a large virtual memory, the reason for performing a garbage collection is mostly to increase locality of reference by eliminating garbage and restructuring, not to prevent the system from running out of storage. This means that there is no instant at which a garbage collection "must" occur. A time that is convenient for the real-time aspects of the computation can be selected for garbage collection. The time for garbage collection can also be chosen by finding a time at which the amount of very short-lived temporary storage is very low. A process only need be interrupted for garbage collection if one of the areas it is using is being garbage collected. It may be possible to write real-time computations which will never be interrupted for garbage collection.

Basic Garbage Collection Algorithm

Garbage collection of area *A* will proceed as follows. First a new area, *A'*, will be created. All the information in *A* will be copied into *A'*. Since a great deal of allocation will occur in *A'* before any freeing occurs, free storage will always be one large block that gradually diminishes in size. Objects are placed next to each other in *A'* if they are sequentially allocated during the garbage collection. The mark and copy phase of the garbage collection begins with the reference to the data base within the area. Each object, *P*, when first encountered in this phase, is copied into *A'* producing *P'* (see Fig. 1). Each object referenced from *P* is garbage collected. References to the new copies of these objects are placed in *P'*. The garbage collection of *P* concludes by returning a reference to *P'*. The second time *P* is encountered in the mark and copy phase a reference to *P'* is immediately returned. When an inter-area link is encountered, it is copied into *A'*, but the inter-area reference is copied directly into the new copy without garbage collecting it. After the main data base has been garbage collected, all the inter-area links into *A* are garbage collected and the links modified so they reference the new objects in *A'*. When this has been completed, all the information pointed at by the main data base or an inter-area link into the area has been copied into *A'*. All fragmentation of storage that may have occurred in *A* has been eliminated and *A'* has been restructured to increase locality of reference. The placement of objects within *A'* has nothing to do with the history of their creation, it is determined by the structure of the data at the time of the garbage collection. When the garbage collection is complete, nothing points into area *A*, so it may be deleted and its address space reused. The problem of multi-area cycles will be covered later.

Similarity to Current Systems

A system with one address space which is broken into areas is analogous to current systems, where the file system provides a single address space (file name and offset within the file). Each file is used for a single purpose, just as an area is. Each file contains much data, while only occasionally containing a single object. Computer systems whose hardware handles a small address space (2**20 or smaller) also have the *job core* for a process that contains the entire address space for that computation. Sometimes these are handled specially, for example by residing on the drum, while files are all on disk. This job core is analogous to the local computation area except the job core must contain all the machine language programs it is using, while the local computation area does not. Usually garbage collection is only done on the job core, but it can be done independently of garbage collection in job cores of other processes. Both the local computation area and the job core have the stack and the heap. Only temporary information is put in the local computation area or the job core. Most information created by a computation is of this nature, however. For a more accurate view of the local computation area, a look at the Multics system is instructive [17]. The local computation area corresponds to all information in the process

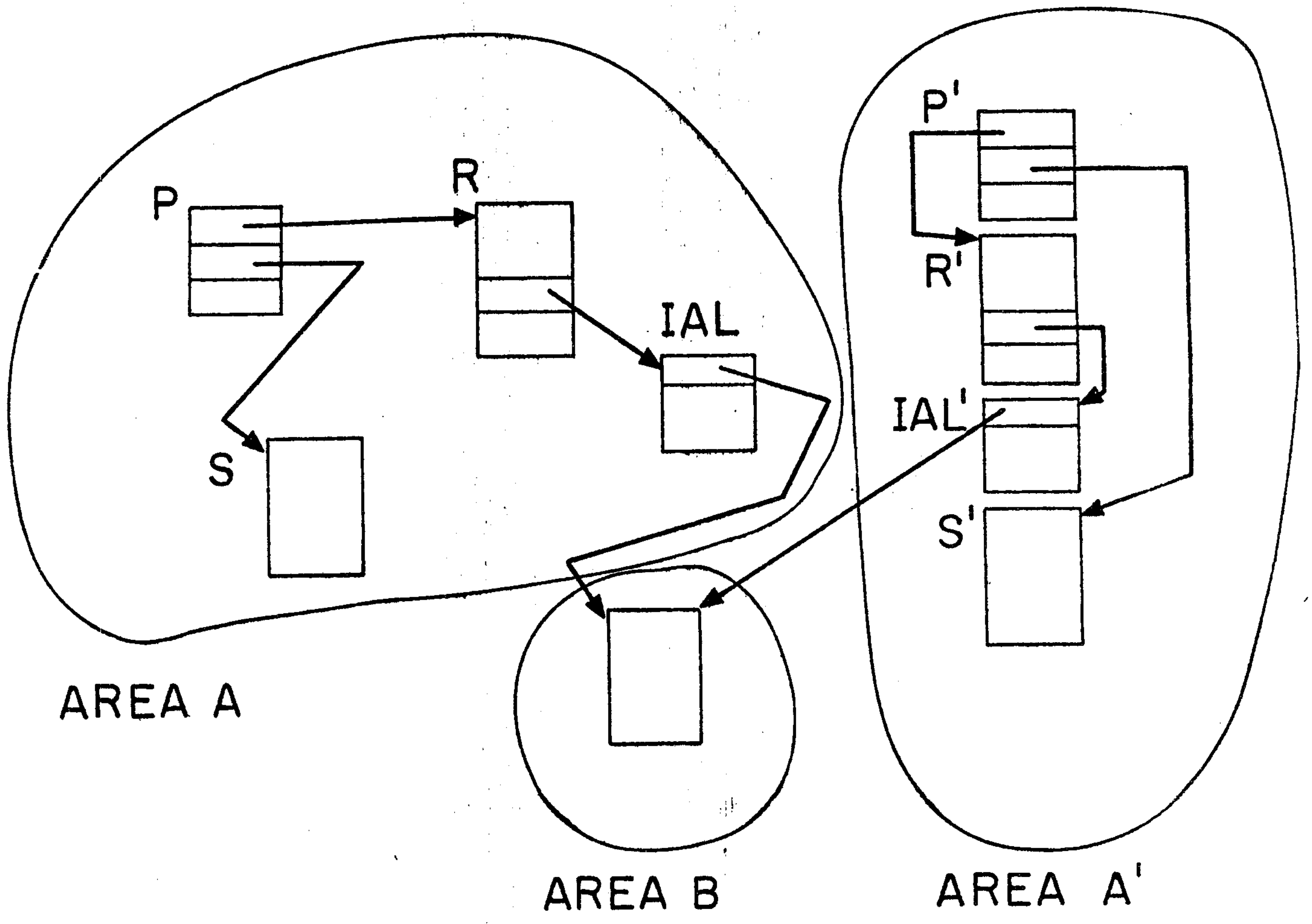

Fig. 1
Garbage collection of area A.

directory on Multics except that the process directory contains some information that supports the multiple address spaces. This information would not be needed on a single address space system.

Systems that have small address spaces can only deal with files by copying them into the job core in pieces and, if necessary, writing them out. This process of creating a new file and then deleting the old one is very similar to a copying garbage collection of an area. This means that some operating systems have prohibited making any changes in a file without performing a copying garbage collection of the file. This fact has allowed many files to be organized so they do not appear to be areas because the "free storage list" is non-existent. Multics does allow the programmer to manipulate files by allowing the files to appear in the address space. Some files on Multics do have free storage lists and can have small changes made to them without too much work. Even here, however, garbage collection of these files is usually done by completely regenerating the file, sometimes by first printing it all out and then reading it all back in. We see, then, that use of copying garbage collection is widespread and should not cause any problems. The fact that some of the operations actually performed on files were hidden by the need to copy them into the job core has obscured the area-like nature of files.

Reliable Maintenance of Lists

We have now seen that the suggestion of taking a very large, single address space and breaking it into areas is similar to what is done on many operating systems that run LISP subsystems. These systems do not allow references from one job core or file to another. The single address space allows objects in different areas to be referenced while the inter-area links allow the address space to be managed regardless of its size. The practicality of the approach depends upon the ability to maintain the lists of inter-area links reliably and quickly. There are basically two approaches that can be taken to this problem: a)force the programmer to do it and b)design the machine language to do it automatically. The first approach has severe reliability problems. It implies the programmer is able to write code that does not maintain the lists of inter-area links properly. Errors in garbage collection are typically non-reproducible intermittent errors that are only detected long after the real error has occurred. In this case the real error would be maintaining lists of inter-area links incorrectly. The error would have no effect until the next garbage collection, when an inter-area reference would be handled incorrectly, resulting in a pointer somewhere that does not point at the structure it is supposed to point at. This error would not be detected until that pointer is used following the garbage collection, if then. By the time this error has been detected, its cause is untraceable. These bugs would be almost impossible to find, resulting in a very unreliable system.

The alternative to making the programmer responsible for the inter-area links is to make the system designer responsible for them. The programmer can be given a machine language that automatically maintains the lists of inter-area links. He is then largely unaware of the maintenance of these lists, thereby making the job of programming easier than above. There is no longer a possibility of error, especially if the machine language is proven to maintain the lists correctly. The problem is to determine when an inter-area reference is being created, and then updating the relevant lists. In order to do this, it is necessary for the machine language to always correctly know when it is dealing with an address and when it is dealing with other information. Most current machine languages do not have this property.

Capabilities

Operating systems that implement protection by the use of capabilities do have this property [6] [24]. On such systems every capability contains information that identifies what it points at and specifies what operations can be performed on it. Capabilities can only be manipulated in restricted ways by the machine language and nothing else can be used as an address. The fact that this mechanism implements protection tells us that *every* pointer is controlled in this way. There are many high level languages that also use data that contain type codes, suggesting that this technique may result in a system that is easier to program. In high level languages the term *capability* is not used. The type code and address of an object should be called an *object reference* since it refers to the object. A word that contains an object reference can contain a reference to *any* kind of object. Wulf [24] tells us that "HYDRA supports *references to objects* (called *capabilities*)."

The approach taken here is to use object references for all objects, including integers, not just for large expensive objects such as files and directories. The fact that object references are used for protection guarantees that all data is correctly identified and that the programmer cannot write any code that will cause data to be incorrectly identified. All machine instructions would take operands that are object references. Since this hardware helps handle object references, it should be designed so it can be used for all purposes that take advantage of object references.

SESAME Systems

A system that requires objects to be manipulated only by object references can be said to have *self-evident data*. That is, the meaning and representation of an object can be discovered merely by looking at the object.

The existence of a system of data types immediately raises the problem of how to add new data types to the system. It must also be possible to add new operations on old data types. A system is *modularly extensible* if both kinds of additions can be made merely by adding a new module without rewriting any other modules.

A large class of computer systems can satisfy the requirements we have been mentioning. These are Self-Evident data, Single Address space, Modularly Extensible

(SESAME) computer systems. SESAME systems may also have multiple processors and will have several, perhaps many, users. SESAME systems will only be practical when they can run fast enough to compete with conventional computer systems.

Data Types

The term "data type" means different things to different people. The high level language designer sometimes uses this term to refer to the "kind" of object, e.g. integers. The integers can be defined by a set of axioms similar to the Peano postulates which define the natural numbers. These axioms define what an integer is by defining a few operations that can be performed on integers and then specifying the properties of the answers returned by these operations. Anything that satisfies these axioms is an integer. The language implementor has a different view of a "data type", especially on a system with self-evident data. In this case a data type includes not only the specification of the abstract object that is being defined but also includes exactly how the operations are implemented so they actually return answers that have the required properties. This is the information specified in the data type field of an object reference: what operations are defined on the object and exactly how they are implemented. Since this report is written from the point of view of a system implementor, the term "data type" will be used in this second sense. The first meaning will be referred to by the term "abstract data type". If each operation has a specification of the properties of its operands and results, then an abstract data type can be specified just by identifying the set of operations that can operate on objects of that abstract type.

High Level and Low Level Abstractions

Some operations on objects can be defined completely in terms of other operations that are already defined on the abstract data type. I will call these "non-primitive" operations. For example, the factorial function can be defined in terms of multiplication, subtraction, and comparison. "Primitive" operations are those that are not defined completely in terms of other operations. These operations must be defined in terms of the implementation of the object. This is done by converting the object reference to a new object with new operations that can be performed on it. This is similar to the amplification of rights described by A. Jones [11] and the use of the *rep description* in CLU [13]. The new object is the implementation of the object from which it was derived. The set of operations on this object should allow the bits in its representation to be modified. The operations would be loads and stores of bit-strings and object references within the object. The meaning of these operations would be totally dependent on the implementation of the object. This new object is the low level abstraction of the high level object from which it was derived. The low level abstraction views the object as existing in a von Neumann machine, while the high level object has only the operations defined on it that are valid on the abstract data type of the object.

Protection

Since object references are used for protection on a SESAME system, the ability to manipulate object references must be limited to meet some basic requirements of capability systems. One basic requirement of a capability system is that object references cannot be manipulated as bit strings to yield new object references, but object references must be stored in memory, as are bit strings. If we can design the computer hardware, we could reserve a bit in each word of memory which would specify whether that word contained an object reference or not. Bit string operations could not be performed on a word that contained an object reference without first turning off the object reference bit and putting zeroes into the word. The object reference bit would be almost invisible to the machine language, since it could not be manipulated directly. Object references could only be created by copying existing object references and by performing legitimate operations on an object reference. Fabry [6] calls this the *tagged* approach.

Another basic requirement of a capability system is that, given only an object reference for one object, it must be impossible to access storage in another unrelated object. This can be achieved if the low level object reference contains the size of the block of storage for that object and if this size is checked on every memory access using that object reference. Remember that high level object references cannot be used directly to access any storage; they must first be converted to their related low level objects. The protection of this kind of system is based on the fact that the only operations that can be performed on an object are those that are defined. It is possible to create objects with arbitrary sets of operations defined on them.

A third basic requirement of a capability system is that, given two object references for which there is some storage overlap, it is essential that the two objects be related in some way and the definition of each object must have knowledge of the possible overlap procedurally embedded. Some procedural knowledge of a fact or property (relationship) is embedded in a piece of code if the code must be rewritten (or at least checked over again) when the fact or property changes. Once the precautions mentioned above have been taken care of, there are still two ways in which the requirement of no unforseen overlap can be violated. A storage allocator can allocate the same piece of storage twice. Any storage allocator that obviously did this would be considered to have a bug, but the entire protection system depends upon the absence of such bugs. This problem can always be solved by writing all storage allocators in the protection kernel of the system.

If storage is re-used, there is a possibility of a piece of storage being freed while there are still object references to the old object. The storage allocator would then re-allocate the storage while an object reference to the old object still existed, thereby violating this basic requirement. This second problem is due to the *dangling reference* problem.

Dangling Reference Problem

The dangling reference problem [14] occurs when an object reference exists after the object has been destroyed (the storage freed). It violates protection if it cannot be detected and especially if the storage for the object is reused. There are two ways to attack this problem. The first way [24] [6] [14] is to never reuse addresses. In this scheme, object references do not contain addresses, but "unique codes". The amount of storage associated with a unique code can be adjusted, and is referenced by an offset that is used with the unique code. The unique code is therefore a segment number. An address consists of a segment number and an offset. Multics appears to be this kind of system but is not because it allows the programmer to convert arbitrary bit strings into segment numbers and is able to reuse segment numbers. Lomet does not reuse addresses, but does not use segments, either. When his area identifier (AID) is considered to be part of the address it is easily seen that his scheme does not reuse addresses. This approach is not taken here. Instead the dangling reference problem is completely eliminated. The system can be designed so that the storage for an object is only freed when there are no more references to the object. Since the entire protection for the whole system depends on this, storage can only be freed after a proof has been performed that there are no more references to it.

There are basically two different ways of performing such a proof. One way is to keep a count of the number of references to the object and when this number reaches zero, free its storage. This is the reference count method. Another way is to look at all the references in the whole system and if none refer to this object, free it. This is called garbage collection. We have already seen that we intend to break the address space into areas and keep lists of all inter-area references in the system. This way the whole system can be searched without actually looking at it all.

Note that use of a "free" statement (explicit freeing of storage) cannot be used since it does not involve performing the proof that there are no remaining references to the object. If such a statement did exist, it would have to find all references to the object and destroy these references before freeing the storage.

Continuations

The high level language LISP is an example of a language with self-evident data. Many LISP implementations, including MACLISP [16], have a particular kind of object that is treated specially: stack frames [25] [3] [6]. This is the only kind of object in MACLISP that is not destroyed by garbage collection, but may be destroyed by other means (such as returning from a function). MACLISP does not really treat stack frames as objects. This results in placing restrictions on the kinds of control structures that can easily be used in MACLISP. This problem arises from a desire to use the standard subroutine call instruction on the machine. This instruction usually places the address of the next instruction in a register, thereby passing it as an argument, and ignores the stack frame altogether. Of course this return location cannot be used without the correct stack frame, so conventions are made, usually requiring this stack frame to be on the end of the stack.

These problems can be solved if the place to return to is made into an object that is referred to with an object reference. The location of the next instruction and the stack frame to use with it will both be contained in the object. Probably the object reference will point at the stack frame which will contain the return location. Instead of having conventions about the format of a stack frame, this will be an abstract data type on which several operations will be defined. One of these operations will be "return". There may be other operations primarily used by the debugger. This abstract type has come to be called a *continuation* in the literature [9]. It is related to the continuations of Reynolds [19]. When a module finishes an operation, it returns the answer, if any, to the continuation passed in the argument list. This also insures that the machine language is aware of the existence and meaning of pointers to stack frames and instruction locations.

If a stack frame is an object, then its storage can only be freed if there are no references to it. Fortunately, in most cases there are no references to a stack frame when it is returned from. The stack algorithm, which allocates and frees storage very quickly, must be able to free such stack frames. Using reference counts on stack frames would allow this to happen. In order for the suggestions in this paper to be practical it must be practical to use a spaghetti stack for the call-save-return stack. This would allow a stack frame to be retained if its reference count is greater than zero when it is returned from. This paper will assume that reference counts are used for stack frames and that a spaghetti stack algorithm [5] is used. Describing these algorithms is beyond the scope of this paper.

Representation of Inter-area Links

Once we have a system on which all the data is identified, it is possible to consider operations in the machine language that will automatically maintain lists of inter-area links. One of the first problems encountered is how to represent an inter-area link. There are basically two approaches. One is to have a special data type for inter-area link that contains the inter-area object reference and some references for chaining the lists of inter-area links together (see Fig. 2). When an inter-area link is created the inter-area object reference is placed inside the link and the object reference for the inter-area link is used in the data. The other approach does not interfere with the data at all, it merely maintains lists that point at all the locations that contain inter-area object references (see Fig. 3). In either case, each link must appear on two lists, 1) a list of links pointing out of the area the link points out of and 2) a list of links pointing into the area the link points into (see Fig. 4). The representation of an area includes: 1) the data base contained in the area, 2) the list of inter-area links leaving the area, 3) the list of inter-area links pointing into the area, 4) the address space covered by the area, 5) the free storage list, and 6) locks (semaphores) to prevent incorrect usage by parallel processes.

The disadvantage of the second method is that it requires that the hardware detect not only the creation of an inter-area link, but also its destruction. The first method at least makes it easier to detect the destruction of an inter-area reference. It is not essential that the

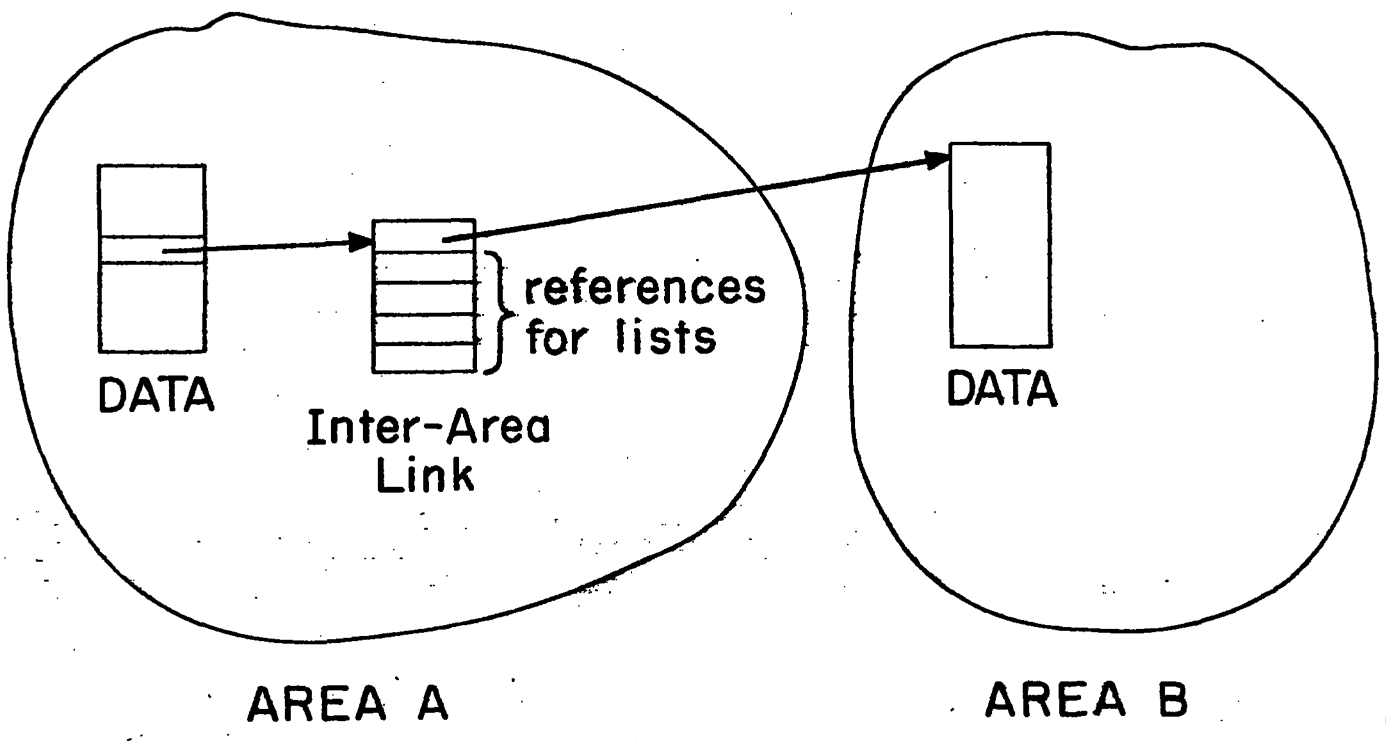

Fig. 2
Representation of an inter-area link - method 1.

destruction of the first kind of inter-area link be detected until the area is garbage collected. A reference count could be used with the inter-area link. The first method is also superior if the same inter-area reference is placed in many locations within an area since it is possible to handle them all with a single inter-area link. It will be assumed that the first method is used.

Maintaining Lists

In order to design the machine language to automatically maintain the lists of inter-area links it is necessary for each instruction to create inter-area links for any inter-area references that are created by the instruction. Machine instructions can be viewed as having two pertinent sub-operations: load and store. Each of these operations has the possibility of creating an inter-area reference. I propose that each load and store operation be checked to see if it does create an inter-area reference and, if so, an inter-area link be created.

Place Area Identification in Page Map

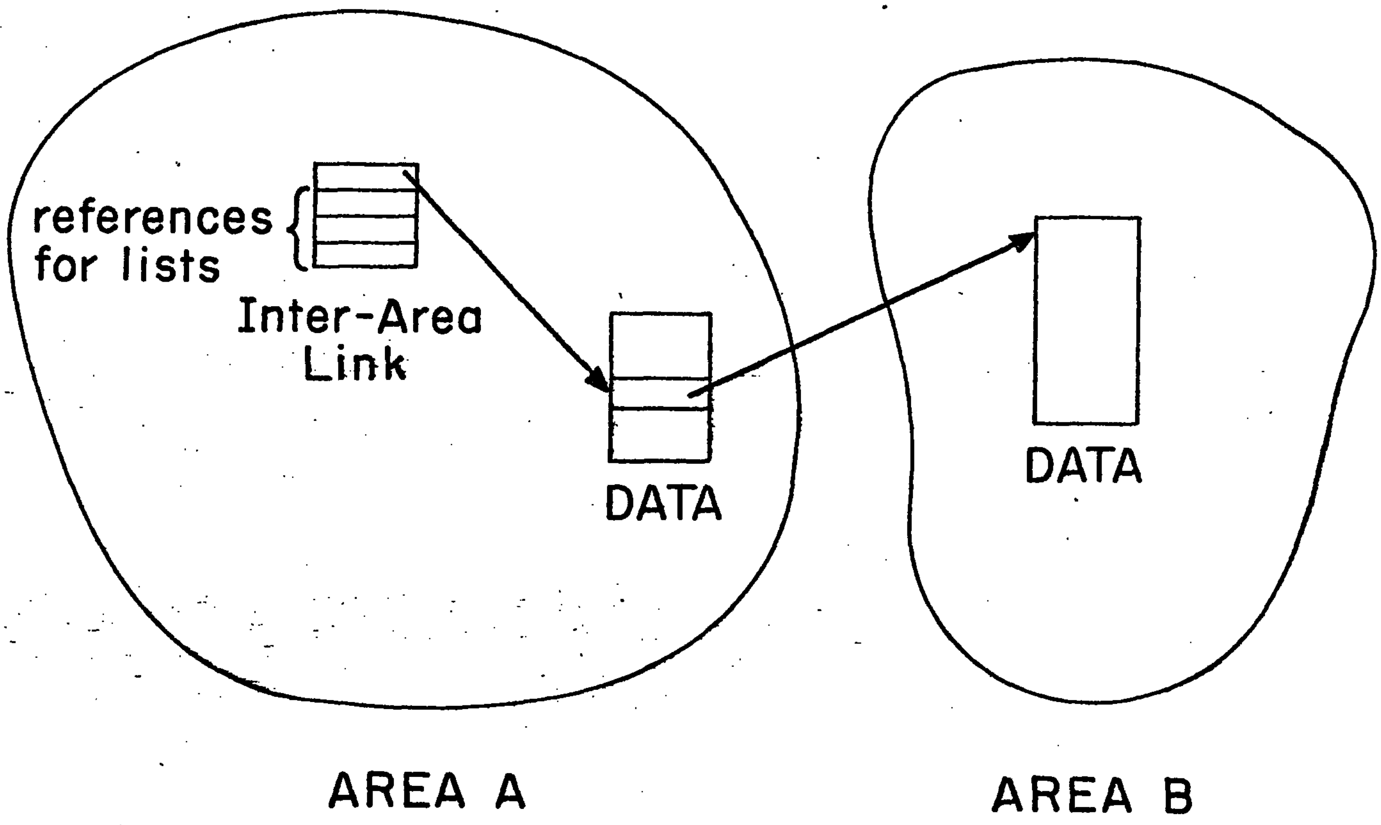

Fig. 3
Representation of an inter-area link - method 2.

Whenever an object reference is placed in a new location, it must be determined what area the location resides in and what area the object resides in. If they are the same, no inter-area link is needed. If they are different, an inter-area link must be constructed. This requires that the operation of finding what area an address is part of be a very fast hardware operation. This could be done by placing this information in the virtual memory page map. Hardware already exists for accessing this very quickly. The only problem with this would be the necessity for an area to contain a whole number of pages. This is not a severe penalty. Multics already requires that each segment contain a whole number of pages. Many systems have similar requirements for files to make the job of allocation of disk easier.

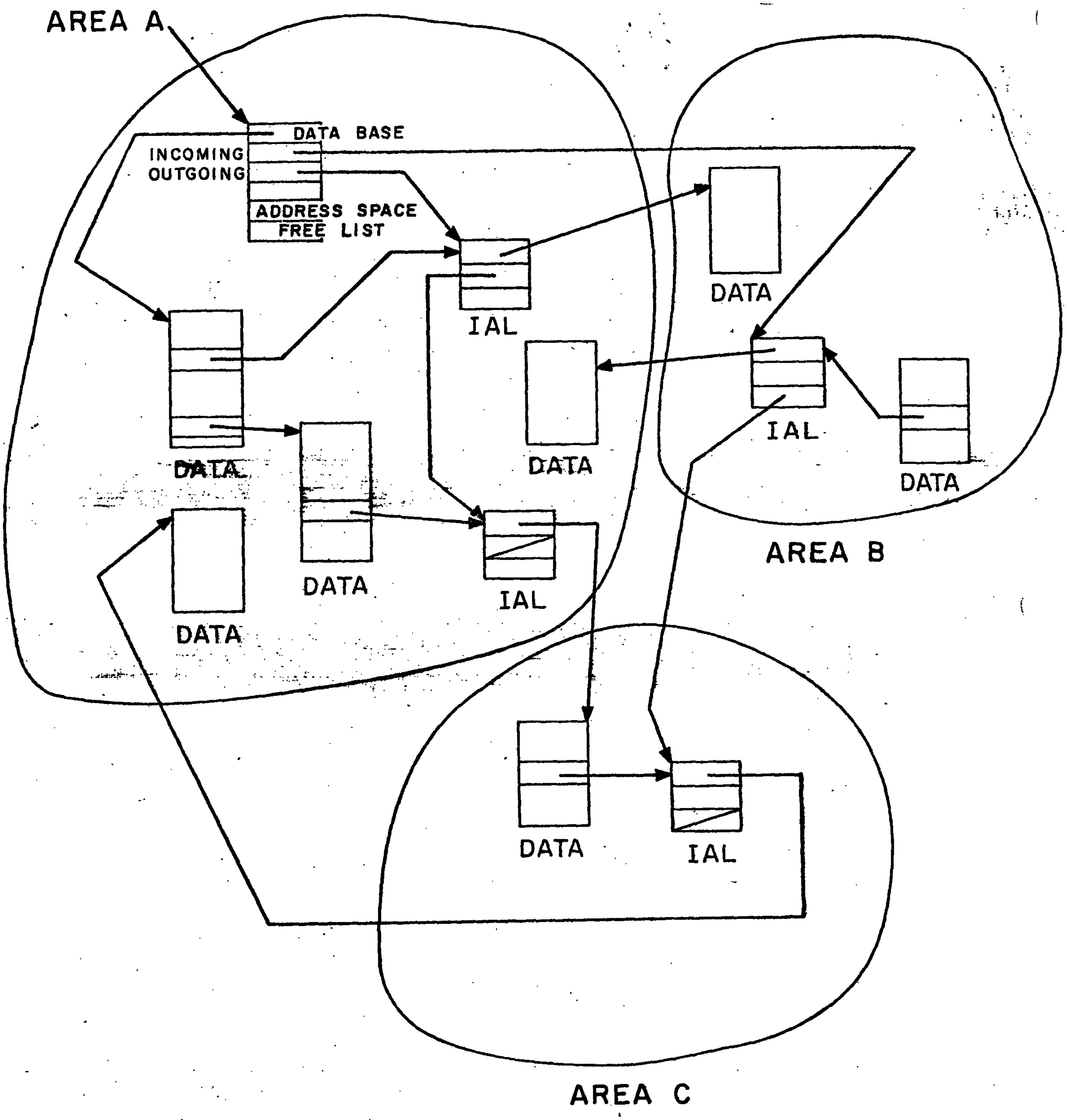

Fig. 3
The lists of inter-area links and the representation of an area.

As described so far, the algorithm has a large overhead. This can be reduced somewhat when we realize that the registers are considered to be part of the local computation area and that moving an object reference from one location to another within the same area cannot require the creation of another inter-area link. This means that loads and stores to the local computation area need not be checked further. Since the page map for the location is accessed during the load or store, the area in which the location resides could easily have been retrieved at the same time, requiring no extra time to perform this check. Computation that stays within the local computation area will run just as fast as on a system that does not maintain lists of inter-area links.

Directly Referencing Links

We have already mentioned that the local computation area does not contain machine code. This means that the instruction counter is an inter-area reference. Another source of overhead with this scheme is the requirement that machine instructions be able to handle inter-area links as if they did not exist, which adds complication to the hardware and/or micro-code. Also, as soon as one thinks of computing on objects in another area, such as searching a list in another area, it becomes immediately apparent that the number of inter-area links from the local computation area to other areas could easily become huge, causing much overhead in creating and destroying these links and in threading and unthreading them from the lists of inter-area links. This could all be eliminated by allowing another kind of link that allows any location in area *A* to directly reference any location in area *B* without using an inter-area link. This will be called a directly referencing link from area *A* to area *B*. If such a link exists from *A* to *B*, we will write $A \mapsto B$. We could eliminate inter-area links from the local computation area altogether by requiring the local computation area to directly reference all areas that it may have an object reference to.

Load and Store

Let us now summarize the load and store algorithms:

Load: Perform the memory access and find the area from which we are loading. If we are loading from the local computation area, nothing further need be done. If we are loading from an area other than the local computation area, check to see if we have loaded an inter-area link. If so, access the true object reference and continue with this. If not, continue with the object reference loaded. Access the page map of the address and find out what area is referenced. If it is the local computation area, we are done. Check the list of areas that can be directly referenced from the local computation area. If the area referenced is one of these, we are done. Otherwise add the area to the list of areas that can be directly referenced from the local computation area.

Store: Find the area in which the object reference is being stored. This is area *A*. If *A* is the local computation area, we are done. If *A* is not the local computation area, find the area referenced. This is area B. If area A directly references area B, we are done. Otherwise create an inter-area link in area A to area B, place the object reference in it, thread it onto the lists of inter-area links, and store the object reference to the inter-area link in the location being stored into.

There is one small improvement that can be made to this algorithm. If the local computation area, L, contains no inter-area links because it directly references all areas that it may have references to, and if we cause the directly referencing relation to be transitive, then no check need be made on any load unless an inter-area link is loaded. Since L directly references all areas it has object references to, the location loaded from, f, must reside in an area, Af, that is directly referenced from L. If the object reference loaded is not an inter-area link, it must reference an area that is directly referenced from Af. Let B be the set of areas directly referenced from Af. Naturally, $Af \in B$. Let M be the set of areas directly referenced from L. If the directly referencing relation is transitive, then if $L \mapsto Af$ and $Af \mapsto X$, then $L \mapsto X$, and $B \subseteq M$. The object reference placed in L need not be checked to make sure it references an area in M. This greatly simplifies the load operation.

Revised Load and Store

Load: Perform the memory access and look at the object reference loaded. If it is not an inter-area link, complete the load immediately. If it is an inter-area link, get the object reference out of it, and find out what area it is pointing into, call it area X. If area X cannot be directly referenced from the local computation area, L, add area X to the set of areas directly referenced from L. Also add all the areas directly referenced by X to the areas directly referenced from L. Complete the load operation.

Store: The store operation is unchanged.

Note that there is no need to check loads from or stores to the local computation area. Instructions that move object references from one register to another need not be checked. The local computation area could be defined as that area containing the current stack frame. If so, instructions that only load from or store to the current stack frame need not be checked either.

Garbage Collection of One Area

The resultant load and store operations are not much more complicated than normal load and store operations. Only rarely are things added to the various lists. There are now two sets of lists. One lists individual inter-area links and the other lists the directly referencing relation between areas. If $A \mapsto B$, then in order to garbage collect area B it is also necessary to garbage collect area A, although area A can be garbage collected separately from B. Consider what would happen in the case of a local computation area, L. It must be remembered there is a strong analogy between the contents of the local computation area and the contents of the process directory on Multics. On Multics, no user can read anything in the process directory of another user. This suggests that one local computation area will probably not reference another one. Computation will proceed, as on Multics, with the user running the command processor in *L*, causing *L* to compute with one subsystem and then another as the user moves between, say, the editor and the compiler. While editing, everything associated with editing will be referenced and placed on the list of areas directly referenced from *L*. While compiling, the same will happen with areas associated with compilation. All during this time the vast majority of storage allocation has been done in *L*, so it may become necessary to garbage collect *L*. *L* directly references many areas, but is not directly referenced by any areas, so it may be garbage collected separately from everything else. During the garbage collection, everything in area *L* is copied into a new area, *L'*. Any references to objects in *L* by way of inter-area links are modified to reference the corresponding objects in *L'*. The areas directly referenced by *L'* are only those areas in the transitive closure of the directly referencing relation from those areas actually referenced by *L'*. If the garbage collection did not occur during compilation, there would probably be no references to the compiler left in *L* and therefore, although *L* has the compiler areas on its list of areas directly referenced, *L'* would not. An area, *A*, will therefore only be directly referenced by local computation areas that actually touched *A* since the last garbage collection of the local computation area. In other words, when a data base is garbage collected, it may be necessary to garbage collect some other processes as well, but only those processes that are using the data base in question.

Implicit in the above analysis is the idea that when the garbage collection is complete, area *L* is deleted. This does not cause problems because all inter-area links that referenced *L* were modified during the garbage collection to reference *L'*. It was assumed that no areas directly reference *L*, so the only references to *L* are within *L*. By destroying the entire area, we destroy all remaining references to objects in *L* and may therefore reuse the address space associated with *L*. Since all references in *L* are destroyed, the directly referencing relation from *L* to other areas can be removed.

Deleting Areas

Areas are analogous to files on current systems. Deleting a file is a fundamental operation that allows a user to control the amount of storage used. Arbitrary areas can be deleted also. There are basically two ways the deletion could proceed. The first method will be called a *hard delete* since it destroys all the information in the area.

Hard delete: To delete area *A*, first find all inter-area links into *A* and change them so they reference the *deleted* object. Almost any computation with the *deleted* object causes a fault. All the inter-area links to *A* are then removed since they are no longer inter-area references. All the storage associated with the address space assigned to *A* is then removed except for the part holding the directly referencing and inter-area link lists. This destroys all references from *A* to other areas, allowing all the inter-area links from *A* to other areas to be removed and all the directly referencing links from *A* to other areas to be removed. The only remaining references to the address space associated with *A* are from areas that directly reference *A*. The address space associated with *A* cannot be reused until all the areas that directly reference *A* have been destroyed and these directly referencing links removed. This can be speeded up by causing these areas to be garbage collected. Any references to *A* in these areas will be converted to references to the *deleted* object during a garbage collection.

The existence of the system of inter-area links allows another method of deleting an area, called a *soft delete*. It only destroys information not referenced by other areas in the system. The major task in deletion of an area is finding and modifying all references to the area. The soft delete essentially does a garbage collection of the area and all areas that directly reference it so that objects referenced by other areas are moved into those areas. All information not saved in this manner is lost.

Soft Delete: To delete area *A*, initiate a garbage collection involving area *A* and all areas that directly reference *A*. Do not create a new area *A'* for the information in *A*. Do not mark from the data base in *A* to mark from. Instead, garbage collect the areas that directly reference *A*. When a reference to *A* is found from area *B*, mark from that reference placing new objects in *B'*. When the garbage collection of the areas that directly reference *A* has been completed, mark from the inter-area links to *A*. When a link from area *C* is processed any new objects found from that link are placed in *C*. When this garbage collection is over, there will be no inter-area links to *A* and no areas will directly reference *A*, so a hard delete may be performed on *A*. This delete will probably be done as part of the clean-up of the garbage collection that will also delete area *B*, leaving area *B'* in its place.

User's View of the System

Consider how this scheme of areas and inter-area links will appear to the user of the computer system. The user will see a rather conventional-looking system with files (areas). As on Multics, there may be files that contain compiled machine code that can be executed directly by the machine without first performing a linkage edit to create a core image. None of the files will be core images. A data base will be a single file that can be used directly by several parallel processes. The user will usually deal with files from his programs by merely accessing their data structures. The programmer will not write I/O statements to deal with his files.

Garbage collection is done by a program in the system libraries, and may be performed on any area. Garbage collection of an area may be initiated by a program or by the user. An area should only be garbage collected if it is likely to contain garbage. The amount of storage allocated in the area since the last garbage collection will give some indication of the need for garbage collection.

The user will also be able to add directly referencing links between areas. This will only be done very rarely, however, so the naive user can easily be ignorant of this possibility. The subsystem designer will be aware of this and may write code that automatically sets directly referencing links at appropriate times. This will eliminate the need for the ordinary user to be aware of directly referencing links.

Perhaps the most annoying thing about this system is the necessity to specify in which area each object is to be placed. This must already be done on Multics. PL/I has always had the concept of areas and has an *allocate* statement that allows the programmer to specify which area to allocate from. LISP is a widely used language that is much more useful than PL/I for most symbol manipulation problems [3]. LISP does not have the concept of areas nor does it require an area to be specified when allocating storage (CONS). On the other hand, LISP systems run entirely in what corresponds to the local computation area. Since all temporary storage should be allocated in the local computation area, an allocate statement that is not given an area as an argument should allocate the object in the local computation area.

Automatic Placement of Objects

Regardless of how carefully an object is placed in the most appropriate area, it is always possible for the data structure to change so an object no longer resides in the most appropriate area. Automatic detection and correction of such situations would relieve the programmer of much work and would decrease the necessity of careful placement in the first place, making the default proposed above very attractive. The soft delete algorithm given above automatically relocates data into other, more appropriate areas. This same idea can be used to move objects from one area to another.

An object is probably in the wrong area if it is not marked when the data structure in that area is traced. A garbage collection of area A could proceed as follows: the data structure inside the area could be traced and copied into the new area A'. The inter-area references must now be found and updated. If any of these inter-area references find an object in area A that has not been marked, this object is moved to the area from which it was referenced, as described in the soft delete algorithm.

The effect of this technique of automatic placement is to leave an object in the area in which it was initially placed until it is no longer referenced from the data base in that area, then it is moved to another area that references the object. This will probably work reasonably well in many instances, providing a good default for programmers who do not want to put effort into placement of objects. This works well in conjunction with the default that objects be placed in the local computation area when they are created, since if the object really belongs in a data base somewhere, the local computation area will stop referencing the object and it will then be moved to the appropriate data base.

Garbage Collecting Multi-Area Cycles

One often-cited advantage of garbage collection over reference count schemes is the ability of garbage collection to handle cycles. If a cycle exists completely within one area, it is easily handled by this scheme (see Fig. 5). If a cycle exists partly in one area, A, and partly in another area, B, some problems arise. If the areas do not reference one another directly, then there will be at least two inter-area links, one from A to B, and one from B to A (see Fig. 6). If the cycle really is garbage, these are the only links holding onto the cycle. If either area A or area B has the automatic mover in effect, then half the cycle will be moved to the other area where it will be eliminated by the next garbage collection. If neither area has the automatic mover in effect, then the only way to eliminate the cycle is to cause both areas to be involved in the same garbage collection. In this case, inter-area links between the two areas are not marked from.

Since each area holds information that logically belongs together, it is expected that most cycles will occur entirely in a single area. Most of the time all the objects in an area will be referenced by the data structure in that area. A cycle that extends over many areas should therefore be referenced by something in each area. As these references are destroyed, one by one, for independent reasons, the cycle will restructure itself into fewer areas until it is contained completely in the last area to reference any part of the cycle.

The existence of the automatic mover does cause some complication for the user. Although in many cases it will relieve the burden of specifying where to place things, the user will have to decide whether to use the automatic mover on an area. There are several variants of the automatic mover that could be used as well. For example, a user might want the automatic mover to only move things out of an area, but not move things into it. The mover could be restricted to only move things into an area, but not out of it. The user may want to do all placement manually.

Protection Implications

The ability to move data from one area to another has serious privacy and protection implications that suggest that the individual reference may be a good source of advice as to whether to move an object out of its original area into one that references it. Hopefully some good defaults can be worked out along these lines that will minimize the amount of work that needs to be done to control the automatic mover. On the other hand, the less the automatic mover is allowed to function, the more necessary it will become for the user to manually destroy inter-area links or to cause garbage collections involving several areas at once.

Further Possibilities

In addition to the advantages already cited, the maintenance of inter-area links and the

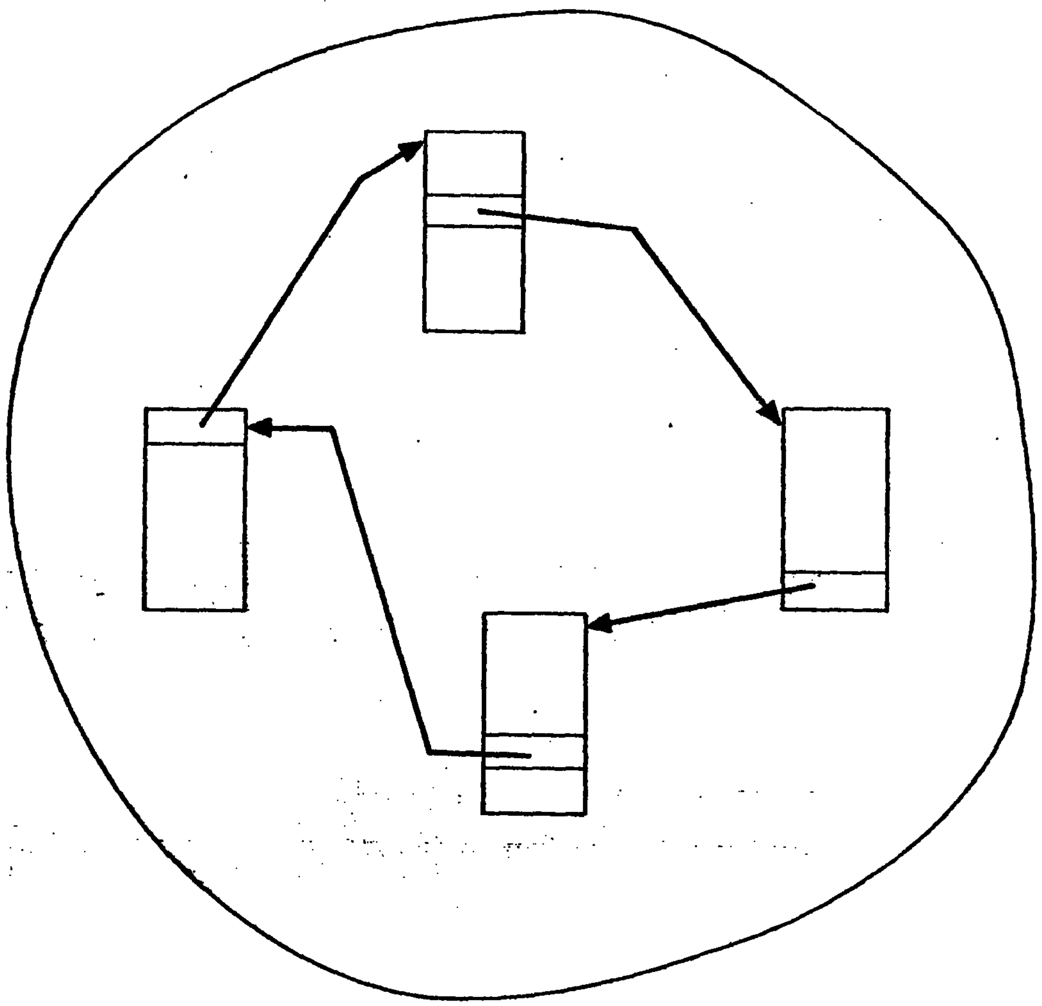

Fig. 5
A cycle within one area.

feasibility of garbage collection means that it is possible to find all the references to an object and modify these references. Algorithms of all kinds that make use of this operation now become feasible. It becomes possible to replace objects with new, restructured versions. For a very low cost, an object can restructure itself during a garbage collection so it will run faster for the operations currently being performed on it. One of the problems with the protection provided by capability systems is that once a highly privileged object reference

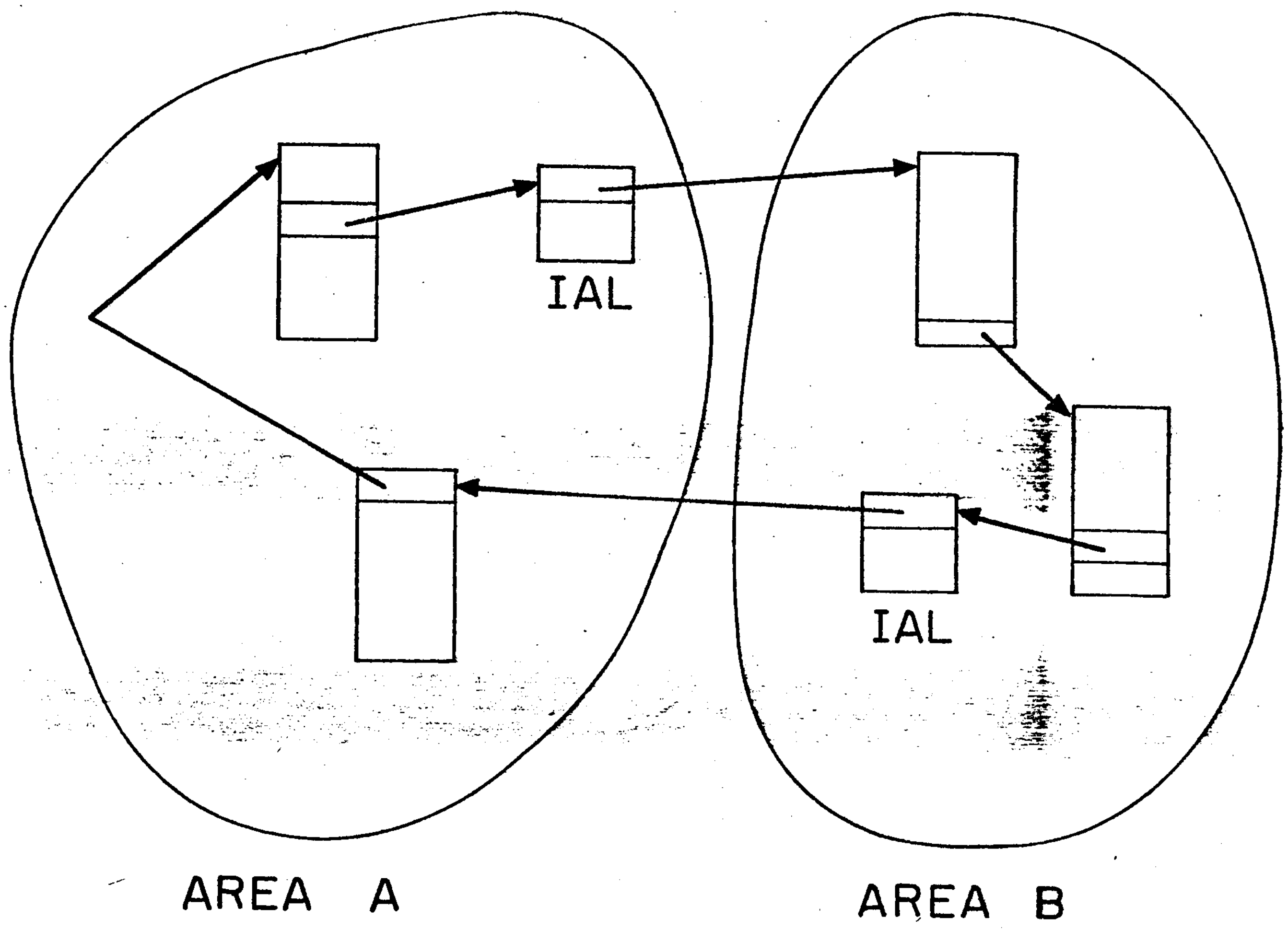

Fig. 6
A cycle covering two areas.

has been given out, it cannot be retrieved. Since all object references to an object can be found and modified, this is no longer true. The area in which the object reference resides gives a strong clue as to whether it is an object reference that may be used improperly or not. These possibilities cause a whole class of algorithms to be practical. Whether they are useful in any particular case should be the subject of future research.

The maintenance of lists of inter-area links is dependent upon the ability to quickly find what area an address is part of. This feature has many other uses as well. It makes it possible to find what area an object resides in. The area could specify the garbage collection algorithm that is to be used in it. The area can supply information that helps to decode the representation for an object. For example, it would be possible to use different formats of object references in different areas. The area would merely contain a procedure that would convert the format of object reference in that area to the format of object reference used by the local computation area. A restriction on this idea is to have certain type codes whose meaning would be area dependent. MACLISP determines most of the type information from the area containing the object. Any object could have some of its information kept by the area it resides in. This could be a good place to find default values for many purposes. These ideas are just some possibilities for the use of this mechanism. Whether they have any merit must await further research.

Conclusion

There has been a tacit assumption by most workers in the field that the dangling reference problem is insoluble and must be lived with. This assumption has been accepted most strongly by designers of operating systems. This paper challenges this assumption and takes the first step in showing it to be false. The mechanisms introduced here do not involve more complexity than the mechanism needed to implement virtual memory. As the remaining steps to complete solution of the dangling reference problem emerge it will be seen that these mechanisms can form the basis for a new computer system that is both fast and easy to use.

Acknowledgements

I would like to thank David D. Redell for his encouragement and for the many discussions I had with him.

References

1. Baecker,H.D. Garbage collection for virtual memory computer systems. *Comm. ACM 15,* 11 (Nov. 1972), 981-986.

2. Baecker,H.D. On a missing mode in ALGOL 68. *Sigplan Notices 7,* 12 (Dec. 1972), 20-30.

3. Berkeley,E.C.,Bobrow,D.G.(Eds). *The Programming Language LISP: Its Operation and Applications.* Information International, MIT Press, Cambridge, 1964.

4. Bobrow,D.G. Storage management in LISP. in *Symbol Manipulation Languages and Techniques,* Bobrow,D.G.(ed), North-Holland Press, Amsterdam, 1968.

5. Bobrow,D.G.,Wegbreit,B. A model and stack implementation of multiple environments. *Comm. ACM 16,* 10 (Oct. 1973), 591-603.

6. Fabry,R.S. Capability-based addressing. *Comm. ACM 17,* 7 (July, 1974), 403-412.

7. Fenichel,R.R.,Yochelson,J.C. A LISP garbage collector for virtual-memory computer systems. *Comm. ACM 12,* 11 (Nov. 1969), 611-612

8. Greenblatt,R. The LISP machine. M.I.T. Artificial Intelligence Working Paper 79, M.I.T., Cambridge, Mass., November 1974.

9. Hewitt,C., Bishop,P., Steiger,R. A universal modular ACTOR formalism for artificial intelligence. Third International Joint Conf. on Artificial Intelligence, Aug. 1973, 235-245.

10. IBM,Corp. *PL/I Language Specifications.* Form GY33-6003-2. International Business Machines Corp., White Plains, N.Y., 1970.

11. Jones,A.K. Protection in programmed systems. PhD Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, June, 1973.

12. Knuth,D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms* Addison-Wesley, Reading, Mass. 1968.

13. Liskov,B.H. and Zilles,S. Programming with abstract data types. *ACM SIGPLAN Notices 9,* 4 (April, 1974) 50-60.

14. Lomet,D.B. Scheme for invalidating references to freed storage. *IBM J. Res. Develop.* (Jan. 1975), 26-35.

15. Minsky,M.L. A LISP garbage collector using serial secondary storage. Artificial Intelligence Memo No. 58(revised), MIT, Cambridge, Dec. 1963.

16. Moon,D.A. *MACLISP Reference Manual.* Project MAC, MIT, Cambridge, April, 1974.

17. Organick,E.I. *The Multics System: An Examination of its Structure.* MIT Press, Cambridge, 1972.

18. Parmelee,R.P., Peterson,T.I., Tillman,C.C., Hatfield,D.J. Virtual storage and virtual machine concepts. *IBM Systems Journal 11,* 2 (1972), 99-130.

19. Reynolds,J.C. Definitional interpreters for higher-order programming languages. *Proc. 25th ACM National Conf.* Boston, Aug. 1972, 717-740.

20. Rochefeld,A. New LISP techniques for a paging environment. *Comm. ACM 14,* 12 (Dec. 1971), 791-795.

21. Ross,D.T. The AED free storage package. *Comm. ACM 10,* 8 (Aug. 1967), 481-492.

22. Steele,G.L.Jr. Multiprocessing compactifying garbage collection. *Comm. ACM 18,* 9

(Sept. 1975), 495-508.

23. Stygar,P. LISP 2 garbage collector specifications. TM-3417/500/00, System Development Corp., 2500 Colorado Ave., Santa Monica, Calif., April 26, 1967.
24. Wulf,W.A.,et al. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM 17*, 6 (June, 1974), 337-345.
25. Yochelson,J.C. Multics LISP. S.B. Thesis, MIT Dept. of Electrical Engineering, June, 1967.