RICE UNIVERSITY COMPUTER PROJECT

Store Management Techniques

J. K. Iliffe

Methods of store management appropriate to machines with tagged addresses are described. It is shown how the information implicit in addresses can be used both by microprogram and system routines to achieve efficient use of register and primary storage.

23 January 1969

Store Management Techniques

J. K. Iliffe*
Rice University

1. The Problem

In order to process information efficiently, it must be made accessible at a rate matching the processor speed. In a stored program computer, it is not usually economical to put all the information required for a calculation in a single physical store: instead, it must be distributed over a number of different "levels", characterised by their access rates. However, the intensity of use of items of information varies immensely in the course of computation, and performance is not seriously impaired if only the most intensely used are supplied at processor speed. To take advantage of this fact, one must arrange to move information into the store level appropriate to its utilization at any instant. Such arrangements may be made by a programmer, given facilities for commanding information transfers. Alternatively, one can apply rules which are independent of any particular program. In time-sharing (and therefore space-sharing) systems, some form of automatic control is essential.

In practice, the following three storage levels are of importance:

(i) Registers, e.g. bistable electronic devices, with access times comparable with the micro-functions of the processor;

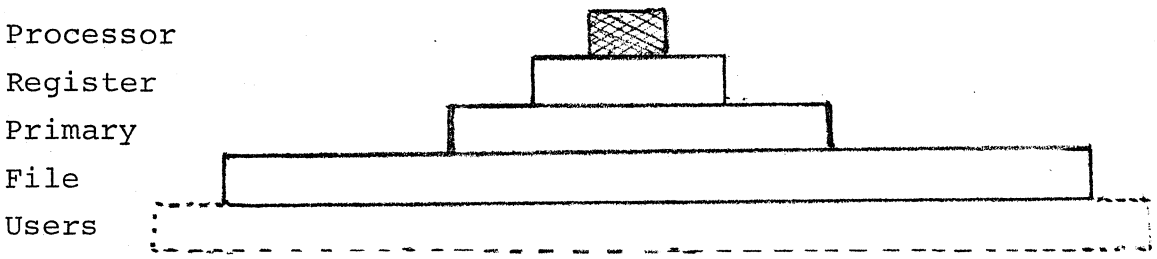
(ii) Primary storage, e.g. magnetic core store, with access comparable with the function speed of the processor;

(iii) File storage, e.g. moving magnetic devices, with mechanically limited access rates.

*On leave of absence from ICL, Stevenage, England.

The storage system of a computer is often thought of as a "hierarchy" with the fastest elements, i.e. the registers, forming the top level. The amount of store at each level is inversely related to its speed, so we can also think of the storage system as a "pyramid", whose top is being transformed by processing devices. More generally, we may regard each level as being "processed" by the combination of active devices and stores which occurs above it. The relative amounts of storage at each level, i.e. the slope of the pyramid, is then determined by applying the matching rule stated above, and as each level is absorbed to form a new "processor", it must be provided in sufficient quantity to match the speed of the next lower stratum. The purpose of the store management rules is then to achieve a given processing rate at the base of the pyramid, where the computer meets the outside world, with minimal amounts of register and primary storage, and processor logic.

Fig. 1. The Store Pyramid



Let us assume that the unit of storage at all levels is a word, whose size is typically about 50 bits. At each storage level, words are assigned to a sequence of physical locations, identified by means of location numbers L_o, L_1, ..., L_k, where k is a parameter to be investigated. As calculation proceeds, the demand for storage at each level is satisfied by finding blocks of consecutive words characterised by the pairs of numbers (L_i, n_i), L_i being the first

word location number, and n_i the number of following words, i.e. the last location of the block is L_{i+n_i}. The blocks so assigned are said to be active. The remainder are inactive: they provide at each level a reserve of space from which to satisfy new demands. When this is impossible, more space must be found by recovering disused space, or by ejecting rarely used blocks to a lower level. It will be seen later (Sec. 4) that the primary store control system works intimately with the process scheduler in regulating the demand for storage.

The movement of information is concealed from programs by causing them to use a set of addresses, whose meaning is independent of the store allocation. In the earliest computers, very little could be done in the way of store management because locations were identified with addresses. Since the acceptance of multiprogramming, crude transformations between address and location numbers, e.g. datrun register and paging schemes, have become common. The relationship between addresses and location numbers can be represented in tabular form. In practice, such a table relates segments (A_i,n_i) of address space to blocks (L_i,n_i) in physical store, with the implication that the j'th element of a segment, $0 \leq j \leq n_i$, is to be found in the jth position of the corresponding block. The table need only relate A_i to L_i, and give the limit value n_i for each segment/block.

There are two ways of using the table, i.e. (A) Unrestricted address computation: If the A_i are indistinguishable from integers, then a table reference must be made whenever an attempt is made to access the store:

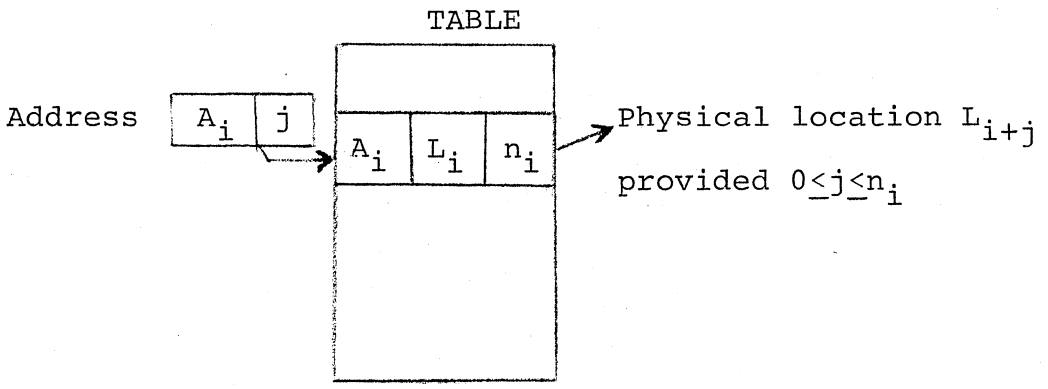


Figure 2: Table reference

In the special cases mentioned above, the paging schemes reduce n_i to a constant (a power of 2), and datum register schemes reduce the table to one entry per program.

In general, provided the relevant parts of the table are stored at a higher level than the information required, this is an acceptable procedure. For example, access to a drum may be controlled through a table in core store. Access to core requires the table (or at least the most intensely used parts of it) to be in registers. Access to registers, on the other hand, is severely penalised because the table itself is often so bulky that it must be accessed by associative methods which, by definition, must be slower than the registers*.

---

*The penalty may still be worth paying if the primary level is much slower than registers, as in the case of slave stores: this can be looked at as a "gain" over primary store or a "loss" compared with register speeds.

(B) Restricted address computation. If addresses are distinguished from other stored items, it is possible to retain the tabular entry as part of the address itself, only referring to the table to update the address when A_i changes:

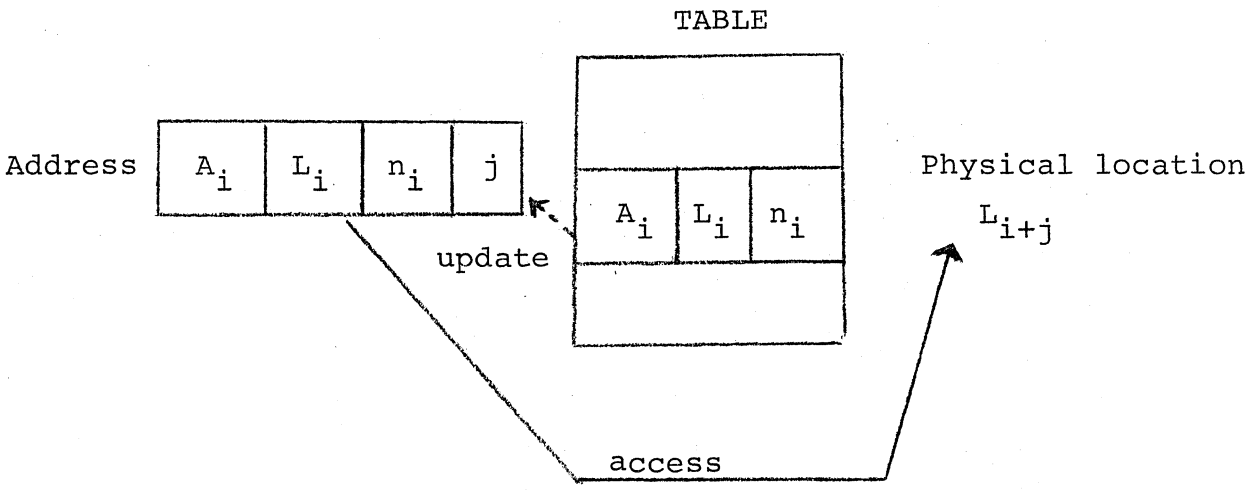


Figure 3: Table by-pass

The advantage of such a method is that reference to physical storage is direct, and a relatively long updating process can be tolerated, even at register level. It depends, just as a paging system does, on the proposition that addresses are used repeatedly to access the same segments of store, although it uses the phenomenon in quite a different way. It should be noted that there is no doubt about the truth of the proposition if the segment is chosen large enough, e.g. a program size; but then the store pyramid is rather uninteresting. The question we have to ask is, whether one can choose such a small (average) segment that the faster levels of storage are minimised, and such that the above proposition remains substantially true. The evidence from paged systems seems to be that for certain classes of problems it is possible, though a fixed segment (page) size can lead to inefficiency in the use of primary storage.

The best known systems of type (B) are the Burroughs B5000 and derivatives. Here, and in the Rice University machine, the Address/Location table is entirely absorbed into stored addresses, with the result that when store is re-allocated, all relevant addresses must be found and changed. A similar technique is used on the experimental Basic Language Machine, though the table (Sec. 4) is also maintained by the primary store control system and could be used as indicated above.

It is quite probable that type (B) systems will be widely used in future. They are inherently faster in store access than type (A), they offer greater flexibility in design, and there are eminently good reasons, outside the scope of these notes, for controlling the formation of addresses (see Ref. 1 Ch. 1). Conventional store management techniques have been examined in practice and by simulation, and reported extensively (for example, see Refs. 2,3). The following Sections are therefore confined to type (B) systems, particularly with reference to register store control (Section 2), and primary store (Section 3,4). Because file access is relatively less frequent, the distinction between the two types of control can be ignored at that level.

2. Register Control

In this Section we consider automatic ways of using register storage to increase computer performance. The methods used must not affect the instruction code of the machine, otherwise a substantial investment is lost; by a similar token, their effect on the processor logic must be minimised.

The normal purpose of registers is to hold numbers or addresses. They may be specialized in use (e.g. a control counter), in which case the sort of information they contain is implicit, or general purpose, accessible by program. In the latter case we presuppose their contents are distinguished by some form of type coding, as in the BLM (Ref. 4). From this point of view, we can see that the registers define a relatively limited domain D in primary storage which is directly accessible by using the addresses they contain. The construction of processor logic, of compiled code, and of hand-coded programs, is normally such that the most immediately required entities, if not themselves in registers, are in D. The most natural way of enhancing performance would therefore be to move D into register storage, and adjust addresses accordingly.

The main objections to the above method are that D is too volatile, and usually too large, for effective maintenance at register speeds. We therefore seek a relatively stable subset D* of D, by rejecting the most rapidly changing addresses (as defined below); and we simplify maintenance by choosing a fixed block size: the size of D* is then limited by the number of registers capable of holding addresses.

As an example, consider a group of 16 general registers, and a register block size of 8 words. Each register can hold either an address (ADDRESS or CONTROL in the BLM sense) or a number. Referring to Figure 4, the register store consists of:

(i) the 16 addressable registers X;

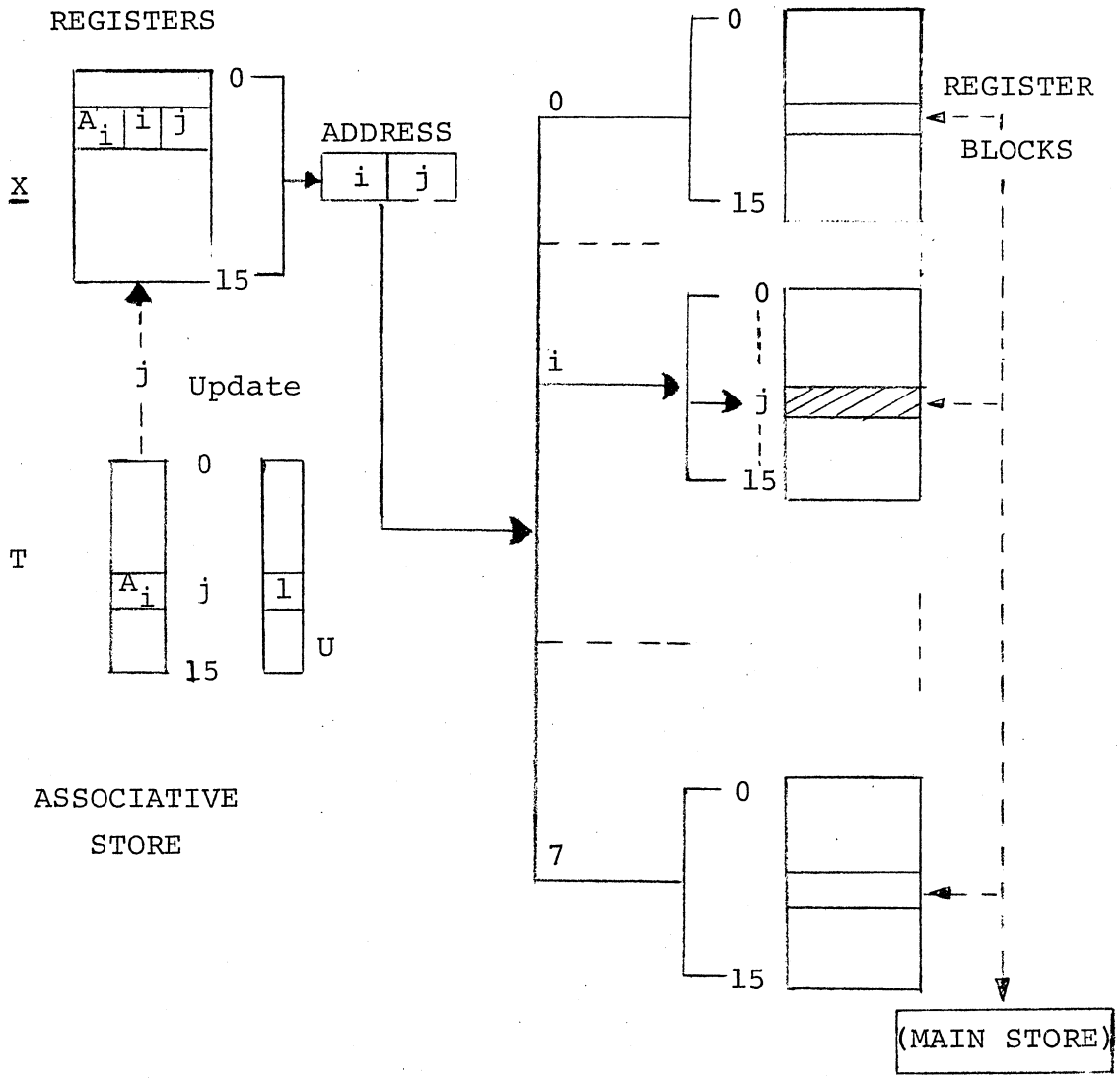


Figure 4: Register Block Organisation

(ii) eight stores of 16 words, forming 16 register blocks (the blocks are spread across separate stores);

(iii) an associative store T which relates segment addresses A_i to the defined register blocks. (T is simply an inverse form of the segment/block table, ordered by block).

and (iv) a use word U whose 16 bits indicate which register blocks are occupied at any time.

If we consider a 20-bit address, its low order 3 bits (i) determine a register store, and the top 17 bits determine, by reference to T, one of the 16 words (j) in that store. The rule of store maintenance are as follows:

1. Loading. When an address is formed in X (e.g. by LOAD in the BLM sense) reference is made to T to determine whether the block it refers to is already loaded. If so, the block address (j) is stored in X. Otherwise, the block is fetched from main store, and a vacant register block is found from U. The block is distributed in the register stores, with appropriate entries in T and U.

2. Ejection. Blocks can be reclaimed if (i) they appear from U to be 'in use', but (ii) there is no address in X corresponding to them. If such a block is protected, the U bit can be inverted directly; otherwise, the block must be returned to primary storage before the space can be re-used. Detection of condition (ii) involves scanning the X registers: this suggests that the "j" field should be held in decoded form in a separate store designed to give such information.

3. Access. There are two cases to consider.

(i) an address is used to access information directly, or in conjunction with a small modifier which does not alter its register block number. In this case, the required information is in register store i, word j.

(ii) otherwise, e.g. when an address is modified outside the block it points to, reference must be made to T to see if the information is in registers. If so, it is retrieved as above;

otherwise, it is obtained from primary storage.

Discussion

In comparing the above system with a conventional slave store, we will assume that reference to T takes the same amount of time as a register fetch. It follows that for 3(i) the proposed configuration can respond to requests in half the time of a comparable slave store, so we must consider how often 3(i) is likely to be the case. Immediate examples of information directly 'covered' by addresses are instruction streams, parameter and local variable lists, top-of-stack areas, and commercial records; it is probable that the register block size could be chosen to cover most references of this sort. It should also be noted that the above rules allow for a register block being brought into use again by rule (1), even if technically disused. Thus, in obeying the Fortran code:

```
      DO  1  K = 1, N
1     A(K) = B(K) + W
```

the register blocks defined by the computed addresses of A(K) and B(K) would normally be available, provided rule 2 is not enforced between iterations.

Conversely, case 3(ii) is most likely to arise from random access to larger arrays, or indirect addressing chains. It is proposed not to load register blocks from such requests: in this sense we are restricting the domain D* to the information which is most likely to be useful.

To summarise the advantages of the register block method, it is expected to be faster than conventional (type (A)) alternatives because:

(a) most successful register accesses avoid intermediate table references;

and (b) addresses are identified as such before they are used, and would therefore initiate earlier access to primary storage.

Also, it is expected to require less storage for a given "hit rate" (the measure of successful accesses to register blocks), since use is made of the X registers in deciding what to eject, and some discrimination can be exercised in loading. However, the net effect on performance remains to be studied by detailed logical design and simulation.

3. Primary Store Characteristics

In descending to the next storage level, we are able to reconsider using the "natural" program structure to provide the basic units of store, so that the active primary blocks are variable in size, and correspond to program segments (A_i, n_i). In this Section, we survey the consequences of such a decision. The main interest centres on the way segments are connected together, and reasons for following certain forms of interconnection over others will be given.

We will refer to any segment (A_i, n_i) by its first word address A_i. A segment A_i is said to be connected to A_j iff in A_i there is an address pointing into A_j. For any segment A_j, let $A_j^!$ be the set of segments connected to A_j. Intersegment connections can be represented in an obvious way by finite, directed graphs. A segment A_i has access to A_j iff there is a directed path from A_i to A_j. Let A_i^* be the set of segments accessible from a given A_i.

The store management system is directly concerned with the evaluation of A_i^* and $A_j^!$ for various segments. At any given time there are just a finite number of bases $B_1, \ldots B_k$ in store, which are segments used by independent processes for access to their programs (see ref. 4). It follows that the active segments must comprise exactly $\bigcup_{i=1}^{k} B_i^*$, since no other information is meaningful. The complementary primary storage is then inactive, and available for re-use. Similarly, if access to A_j has to be monitored, e.g. because of a change in position, then $A_j^!$ must be found, and suitable

changes made in addresses therein, unless the table reference (Figure 2) is retained.

In theory, finding A_i^* or $A_j^!$ involves scanning the entire active store. In practice, three factors drastically reduce the amount of work involved. Firstly, not all blocks can contain connecting addresses. Secondly, the bases from which a given segment is accessible are normally restricted to a small subset of B_i, i=1,...,k (otherwise, process synchronisation becomes a troublesome problem). Thirdly, over a substantial region of store, it is possible to form segments into a tree structure, for which $A_j^!$ consists of exactly one segment, for any A_j. The implications of these reductions will be apparent in Section 4. It remains to consider techniques which can be used when the "natural" segment sizes do not admit economic management.

Paged Systems

A fixed size paging scheme is described in ref. 1, ch. 5. It has the effect of partitioning large segments into separate blocks, allowing any segment to be distributed over primary and file storage, and eliminating the need for reorganisation of primary store. Advantage is taken of the type B addressing mechanism to by-pass page table references, as for the register block scheme. For certain classes of problem, particularly those involving serial access to large arrays, one may then expect the primary store requirement to be reduced, despite the "round-up" effect of the page frames.

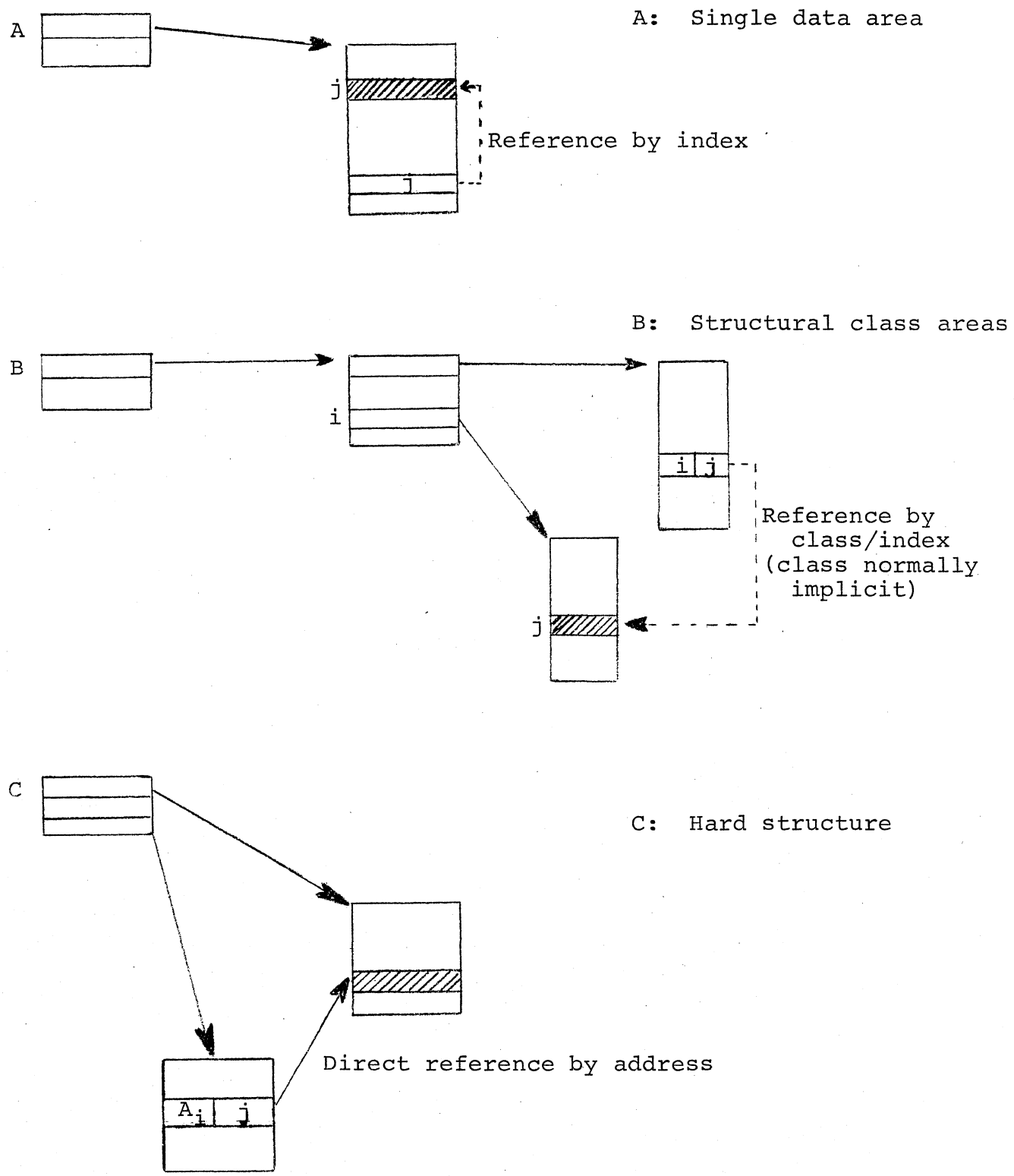
Small Segments

Although, even in a paged system, small segments (e.g. less than page size) can be packed efficiently, significant system overheads may develop from repeated evaluation of A_i^* and $A_j^!$. The obvious extreme is found in list processing systems. It is of interest to find situations in which groups of segments can be

treated as independent subsystems, with specialised storage control rules. One such example is provided by the assembly program, which creates a fixed subsegment structure analogous to the store of a von Neumann machine. Other special schemes can be devised by substituting soft versions of the hard "fetch" and "store" orders, which prevent the propagation of certain classes of address. Chained arrays have been provided in this way on the experimental BLM. Certain compilers have very limited requirements, and advantage can be taken of the fact. At the same time, one must avoid specialisations which limit the efficiency of programs in order to achieve apparently low store control overheads in the "system".

The choice between "hard" and "soft" dynamic store control is illustrated in Figure 5. Program A has just one data block, within which references are made by index values, and store management is entirely A's responsibility, as in conventional machines. In program B, separate blocks are used, e.g. for different classes of data structure. Any single item must be referenced by a pair of indices, though in some languages (ref. 6) the structural class is implicit and would not be stored. Management within each class is carried out by specialised routines in B, but the system now controls the separate classes. In program C, direct reference between segments is allowed, and the system has full responsibility for store allocation. Obviously, C will have greater "overhead" than A, even though it may be doing the same job more efficiently. On the other hand, A and B can be used very effectively for certain classes of problems in which limited formats and access methods are acceptable.

Figure 5: Hard and Soft Structure.



4. Store Control in BLM

To illustrate primary store control techniques, the system developed by J. J. L. Williams for the experimental BLM (ref. 4) will be described in this Section. The Burroughs B5500 system is similar (ref. 5), except as noted below.

Space is requested by entering a system routine, giving the type, size, and number of elements to be provided. Such requests can be normalised immediately into blocks of words, which are then dealt with by the store control routines.

The problem is thus to satisfy a continuous demand for blocks of different length in such a way that those immediately needed (for processing or peripheral transfer) are held in the core store, and so that the control algorithm itself takes as little as possible of otherwise useful time. The solution takes the form of a permanent "store control process", which can apply various recovery procedures, and whose priority can be adjusted in accordance with store congestion. The system is intended to cope with a wide variation in demand, but a mean block size of about 100 words is assumed in this discussion.

In order to simplify the control algorithm, the underlying program structure is constrained to the form of a tree. Other structural relations are superimposed on this by means of individual addresses, but for each active process there is just one segment which can contain addresses: this is called the process base. Up to eight active processes are allowed, and their bases are referred to as "B_i", i being the process number 0,1,...7; in a typical process, the base, which includes the registers, contains 30 - 40 elements.

In the program tree, each element belongs to one of the processes; its owner is therefore identified by a process number. The owner is a hereditary property of the element, in the sense that all the segments in the tree it defines have the same owner; when a process is abolished, all the segments it owns are inactivated.

Hence the primary storage tree can be portitioned uniquely into a set of sub-trees T_i, corresponding to the current processes. If the machine is "empty" there is still one resident process, associated with the operating system: this is identified as process 0, and the corresponding subtree T_0 represents permanent system information.

A process π_i is allowed access only to its own program, T_i, and to parts of the program associated with its parent (i.e. the process which created π_i). Since all processes, directly or indirectly, are created by π_0, T_0 is accessible as shared information. In other words, in B_i the addresses would all point into T_i or T_0, unless π_i was a sub-subprocess of π_0. Conversely, given any A_k, owned by T_j, all connections to A_k will be found in B_j, or in the bases of subprocesses (if any) of π_j, except for the single connection in T_j.

Store Categories

Active store is defined as before as $\bigcup_{i=0}^{7} B_i^*$, if we recognise the principal addresses in T_i as part of B_i. The experimental BLM has a function CLEA which deletes segments A_k from the program tree: however, before the space can be re-used it is necessary to scan A_k' and annul any connections that remain. Until this is done, the deleted segments are classified as cleared: in certain situations A_k' becomes empty, in which case recovery is immediate. The remaining store is inactive or free.

Free Block Chain

Free blocks are chained together in core. A request for N words of store is met by scanning the Free Block Chain (FBC) for the first which is large enough. If successful, the required amount is activated, and the remainder is left on the chain unless it falls below a certain size (currently 8 words), when it is treated as

cleared. When the FBC scan fails, the process asking for store is halted in a state known as "WSTO" (waiting for store), and control passes to another process which is ready to run. The FBC scan is a direct in-line cost to the calling process, and for this reason the mechanism of detachment and activating the new segment has been made as simple as possible, even though the later recovery processes are made slightly more complicated as a result.

Active Block Table

Although the segment/block map is fully absorbed into addresses (Figure 3), it facilitates store recovery to maintain an Active Block Table (ABT), whose entries give the starting location of each block not in FBC (excluding the small "cleared" blocks mentioned above). An entry is made in ABT when a store request is satisfied; when a block is relinquished (cleared) the block itself is marked. Thus ABT summarizes the core disposition at any instant, irrespective of the users' program structure.

4.1 Store Recovery Procedures

The store recovery process is normally of low priority, and is given control only when the remaining processes are suspended, waiting for store or other resources. (The priority can be increased to satisfy an urgent demand for space). Action depends on a group of markers set by other processes in the course of store requests. The most significant of the markers set determines which recovery procedure will be applied.

The way of setting markers, and the recovery procedure they provoke, are as follows (in descending order of significance):

SRP0 : Sort ABT into ascending core store order. The marker is set whenever a store request is satisfied.

SRP1 : Shorten ABT by removing all cleared blocks A_k, scanning A'_k to annul connecting addresses. Regenerate FBC and release all processes in state WSTO. The marker is set whenever the in-line FBC scan fails.

SRP2 : Reorganise the core so as to form larger blocks on FBC. The reorganisation may be local, so as to meet a particular request with minimum delay, or general, moving all active blocks to one end of core. Blocks involved in peripheral transfers are not moved. All connections to moved blocks are updated. The SRP2 marker is set when a process WSTO is not satisfied by SRP1.

SRP3 : Eject all blocks, other than those directly accessed from B_i, i=0,1,...7, to file storage. (This is a reverse application of the principle used in Section 2 to control register loading). The SRP3 marker is set when a process WSTO is not satisfied by SRP2.

If SRP3 fails to meet a store request, overloading is indicated, and steps must be taken to remove one or more processes from the active list. Since this is an expensive operation, an attempt is made to avoid it by requiring each process to specify before being accepted how much core store it needs. The total requirement should not exceed the available core at any time: if it does, SRP3 may fail to satisfy a request, and the offending process would then be de-activated. The core requirement must be chosen to keep the interchange with file store within bounds, as well as to minimise primary storage. The best policy to follow, however, depends on the individual system and work load.

Notes on B5500 DFMCP

From the available literature, the B5500 Disk File Master Control Program appears to be similar to the BLM system structure. The PRT and stack roughly correspond to the storage tree and base

of each process. The main differences are (i) that the PRT structure is one level only, and it can contain multiple references; (ii) system activities in B5500 are not subject to the same protection as users; and (iii) in character mode operations the addresses are not distinguished as such, so that the recovery procedure corresponding to SRP3 is made more difficult. The FBC's correspond. There is no ABT: all blocks (active and inactive) are chained together in core, and adjacent inactive blocks combine as soon as formed. In general, there appears to be more in-line processing to maintain tables and find storage, but SRP0 and SRP1 are avoided, and so is core reorganisation (SRP2). If a request is not satisfied from the FBC, one or more segments are ejected to disk, though it is not clear on what basis they are chosen. There is no independent store recovery process.

A note on the Rice Computer

A FBC on the above lines was used in the first version of the SPIREL System. It should be noted that after several years this was changed in the following way, with appreciable increase in efficiency:

(i) if a store request begins by inactivating a block, and the resulting inactive area is large enough, it is used to satisfy the request;
(ii) otherwise, an attempt is made to satisfy the request from one particular inactive block;
(iii) if (ii) fails, but there is enough inactive space in core, reorganisation (SRP2) is forced, to provide the maximum inactive block, and (ii) repeated.

5. References

1. J. K. Iliffe, "Basic Machine Principles" (Macdonald, London; American Elsevier) 1968.

2. J. S. Liptay, "Structural Aspects of the System/360 Model 85. II: The Cache", IBM Systems Journal Vol 7, No. 1. 1968.

3. M. H. J. Baylis, D. G. Fletcher, D. J. Howarth "Paging Studies Made On the ICT Atlas Computer" IFIP Congress 1968.

4. J. K. Iliffe, "Elements of BLM", Rice University Computer Project, 1968.

5. Burroughs Corporation, "A Narrative Description of the Burroughs B5500 Disk File Master Control Program" 1966.

6. N. Wirth and C. A. R. Hoare, "A Contribution to the Development of Algol", C.A.C.M. Vol 9, No. 6. June 1966.