SRI International

A STORAGE REPRESENTATION FOR EFFICIENT ACCESS TO LARGE MULTIDIMENSIONAL ARRAYS

Technical Note 220

April 1980

By: Lynn H. Quam, Senior Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

SRI Project 1009

The work reported herein was supported by the Defense Advanced Research Projects Agency under Contract No. MDA903-79-C-0588.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

ABSTRACT

This paper addresses problems associated with accessing elements of large multidimensional arrays when the order of access is either unpredictable or is orthogonal to the conventional order of array storage. Large arrays are defined as arrays that are larger than the physical memory immediately available to store them. Such arrays must be accessed either by the virtual memory system of the computer and operating system, or by direct input and output of blocks of the array to a file system. In either case, the direct result of an inappropriate order of reference to the elements of the array is the very time-consuming movement of data between levels in the memory hierarchy, often costing factors of three orders of magnitude in algorithm performance.

The access to elements of large arrays is decomposed into three steps: transforming the subscript values of an n-dimensional array into the element number in a one-dimensional virtual array, mapping the virtual array position to physical memory position, and accessing the array element in physical memory. The virtual-to-physical mapping step is unnecessary on computer systems with sufficiently large virtual address spaces. This paper is primarily concerned with the first step.

A subscript transformation is proposed that solves many of the order-of-access problems associated with conventional array storage. This transformation is based on an additive decomposition of the calculation of element number in the array into the sum of a set of integer functions applied to the set of subscripts as follows:

$$\text{element-number}(i,j,\ldots) = fi(i) + fj(j) + \ldots$$

Choices for the transformation functions that minimize access time to the array elements depend on the characteristics of the computer system's memory hierarchy and the order of accesses to the array elements. It is conjectured that given appropriate models for system and algorithm access characteristics, a pragmatically optimum choice can be made for the subscript transformation functions. In general these

models must be stochastic, but in certain cases deterministic models are possible.

Using tables to evaluate the functions fi and fj makes implementation very efficient with conventional computers. When the array accesses are made in an order inappropriate to conventional array storage order, this scheme requires far less time than for conventional array-accessing schemes; otherwise, accessing times are comparable.

The semantics of a set of procedures for array access, array creation, and the association of arrays with file names is defined. For computer systems with insufficient virtual memory, such as the PDP-10, a software virtual-to-physical mapping scheme is given in Appendix C. Implementations are also given in the appendix for the VAX and PDP-10 series computers to access pixels of large images stored as two-dimensional arrays of n bits per element.

I INTRODUCTION

This paper addresses problems associated with the access of elements of large multidimensional arrays when the order of access is either unpredictable or is orthogonal to the conventional order of array storage. Large arrays are defined as arrays that are larger than the physical memory immediately available to store them. Such arrays must be accessed either by the virtual memory system of the computer and operating system, or by direct input and output of blocks of the array to a file system. In either case, the direct result of an inappropriate order of reference to the elements of the array is the very time-consuming movement of data between levels in the memory hierarchy, often costing factors of three orders of magnitude in algorithm performance.

Access to elements of large arrays is decomposed into three steps: transformation of the subscript values of an n-dimensional array into the element number in a one-dimensional virtual array, mapping of virtual array position to physical memory position, and access to the array element in physical memory. The virtual-to-physical mapping step is unnecessary on computer systems with sufficiently large virtual address spaces. This paper is primarily concerned with the first step.

The subscript transformation conventionally used is a linear combination of the subscript values, of the form:

$$\text{element-number}(i,j,k, \ldots) = a + b*i + c*j + \ldots$$

A more general subscript transformation is proposed that we believe will solve most of the order-of-access problems associated with conventional array storage. This transformation is based on an additive decomposition of the calculation of element number in the array into the sum of a set of integer functions applied to the set of subscripts as follows:

$$\text{element-number}(i,j,k,\ldots) = fi(i) + fj(j) + \ldots$$

II A CASE STUDY: THE STORAGE OF LARGE IMAGES

The storage of large images is a special case of the storage of large arrays. It has been common practice in digital image processing and image analysis to represent images as two-dimensional arrays of pixels stored in conventional order, for example, by rows in a left to right, top to bottom manner referred to as "row order." Algorithms that access the pixels in the same order in which they were stored can efficiently handle even very large images in one pass through an image stored on secondary storage.

There are, however, many image analysis algorithms that do not access the pixels in strict row order. Geometric transformations such as image rotation and transposition have a predictable order of access, but it can be totally orthogonal to the storage order. Such algorithms have deterministic models for accessing order.

As an example, consider a straightforward implementation of array transposition as B[i,j]=A[j,i] applied to a 2048 x 2048 image, which is conventionally stored one pixel per byte. Assume the system uses any page replacement algorithm related to the least-recently-used (LRU) algorithm. Under those conditions, this array transposition calculation would cost a page fault on every pixel if the entire image would not fit into physical memory. On a VAX computer system running either VMS or Berkeley UNIX, with a page size of 1K bytes and a cost of approximately 30 milliseconds per page fault, this task would consume 18 hours, only one minute of which is actual processing. Other systems such as KL-10 computers running TOPS-20 produce comparable results.

The order of access of algorithms such as edge following and region growing is not deterministic, since that order is data dependent, but it can be modeled as a stochastic process. There are mechanisms for implementing many of these algorithms so they access their data in storage order, but usually these mechanisms are far more complex than the image analysis algorithms themselves and introduce other inefficiencies or severe restrictions.

Reddy* and others have proposed that large images be stored not by rows, but by rectangular blocks roughly equal to the size of a virtual memory page. With such a representation, any order of access to the image with a reasonable degree of two-dimensional locale of reference will involve only a small number of pages at a time. For example, assume we have a 2048 x 204 x 8-bit image with a block size of 32 x 32 pixels (1K pixels per block), and a page size of 1K bytes. To get the pixels from any row or column requires access to 64 pages. For an image stored in row order, the rows require access to 2 pages, and the columns require access to 2048 pages. To extract a 128 x 128 window from an image stored by blocks requires access to a minimum of 16 pages and a maximum of 25 pages, depending on the alignment of the window with the 32 x 32 block boundaries. Similarly, to extract the same window from an image stored in row order requires access to a range of from 128 to 256 pages.

The major issues that must be dealt with to achieve widespread acceptance of such a block-oriented image representation are efficiency of access to the pixels, flexibility in the arrangement of the pixels, and ease and transparency of use in a high-level language.

III PROPOSED SUBSCRIPT TRANSFORMATION

The proposed transformation from array subscript values to the storage position in a one-dimensional virtual array is of the form:

$$\text{element-number}(i,j,k, \ldots) = fi(i) + fj(j) + \ldots$$

The following examples illustrate some of the different choices of the transformation functions for two-dimensional arrays:

---

* Raj Reddy and Greg Gill, "Representation Complexity of Image Data Structures," Proceedings: Image Understanding Workshop, pp. 28-30 (May 1978).

Example 1: Storage by row, ni elements per row:

```
element-number(i,j) = i*ni + j

or              fi(i) = i*ni
                fj(j) = j
```

Example 2: Storage by column, nj elements per per column:

```
element-number(i,j) = i + j*nj

or              fi(i) = i
                fj(j) = j*nj
```

Example 3: Storage by block:

Given an array of ni rows and nj columns,it is often useful to divide the array into rectangular blocks which contain bi rows and bj columns. Usually, bi*bj will be chosen to be related to the virtual memory page size, probably in the range of 512 to 2048 bytes, but it is not necessary to require alignment of array blocks with virtual memory pages. (In the representation proposed here, array rows and columns must be padded to exactly fill an integral number of array blocks to enable the additive decomposition of the transformation.)

Assuming that elements are stored in blocks in row order, and that blocks are arranged in row order, the following calculations compute the element number within the array:

```
element-number(i,j) =
                        (j mod bj)
                      + (i mod bi) * bj
                      + (j div bj) * bi*bj
                      + (i div bi) * bi*bj * (nj div bj)

or            fi(i) =   (i mod bi) * bj
                      + (i div bi) * bi*bj * (nj div bj)

              fj(j) =   (j mod bj)
                      + (j div bj) * bi*bj
```

In the proposed accessing scheme, the values of fi and fj are pre-calculated over the ranges of each subscript and stored in tables, rather than calculated on every access to the array.

Example 4: Storage by nested blocks:

A generalization of the block storage scheme is the division of the array into a nested sequence of blocks. At each level k in the sequence, the block consists of bi[k] x bj[k] sub-blocks of level k-1. Block level 0 consists of a single-array element.

Assume that the array has been padded to dimensions ni and nj which have as factors the desired block dimensions:

$$ni = bi[1] * bi[2] * \ldots * bi[nk]$$
$$nj = bj[1] * bj[2] * \ldots * bj[nk]$$

define $pi[0] = 1,\ pi[k] = pi[k-1] * bi[k]$
and $pj[0] = 1,\ pj[k] = pj[k-1] * bj[k]$

then $fi(i) =$

$$\sum_{k=1,nk} ((i \operatorname{div} pi[k-1]) \bmod bi[k]) * pi[k-1] * pj[k]$$

and $fj(j) =$

$$\sum_{k=1,nk} ((j \operatorname{div} pj[k-1]) \bmod bj[k]) * pi[k-1] * pj[k-1]$$

An interesting case of the nested block representation is the binary case, where for all k, bi[k]=bj[k]=2. This subscript to element-number transformation would cause a two-dimensional array to be stored as follows:

```
 0  1  4  5 16 17 20 21  64  65  68  69 ...
 2  3  6  7 18 19 22 23  66  67  70  71 ...
 8  9 12 13 24 25 28 29  72  73  76  77 ...
10 11 14 15 26 27 30 31  74  75  78  79 ...
32 33 36 37 48 49 52 53  96  97 100 101 ...
34 35 38 39 50 51 54 55  98  99 102 103 ...
40 41 44 45 56 57 60 61 104 105 108 109 ...
42 43 46 47 58 59 62 63 106 107 110 111 ...
 .  .  .  .  .  .  .  .   .   .   .   .
 .  .  .  .  .  .  .  .   .   .   .   .
 .  .  .  .  .  .  .  .   .   .   .   .
```

The primary advantage of the binary nested block representation is that it is directionally isotropic in efficiency of access at all scales, from very small array element neighborhoods to large blocks of very large arrays. This arrangement is useful because of the hierarchical structure of computer memory.

IV ORDER OF ACCESS AND MEMORY HIERARCHY

Nearly all computer systems have at least two levels of memory hierarchy: the main memory of the CPU and a disk file system. Many newer computers have an additional level of memory hierarchy that consists of a high-speed cache between the CPU and the main memory. In the near future high-speed page caches built from bubble memory elements are expected. Most of these levels of memory exist because of tradeoffs between cost per bit and access time.

It is conjectured that, given a model for the characteristics of the memory hierarchy together with a model for the order of accesses made to the elements of an array stored in that memory hierarchy, a pragmatically optimum choice can be made for the subscript transformation functions for that array. Future research is needed to explore this conjecture.

For algorithms whose order of access does not depend on data content, the order of access is almost trivial to model. The order of access of the remaining algorithms must be modeled stochastically. For example, the order of access of an edge-following algorithm might be modeled by the statistics of a random walk. However, a complete model for the order of access to the memory is quite difficult in a multiprocess environment.

Memory hierarchy might be adequately modeled by parameterizing each level of the hierarchy in terms of its granularity of storage, transfer latency time, and transfer bandwidth.

In the absence of specific models, a useful strategy for selecting the subscript transformation tables is to provide directionally isotropic time of access to regions of any size. This is important in order to take advantage of characteristics of the memory hierarchy, such as the ability to cluster many pages on the same track of the disk, and many tracks in the same cylinder to minimize head motion. The binary nested block scheme implements this strategy without needing a specific model for the memory hierarchy.

V TRANSFORMATIONS VIA ACCESS TABLE MODIFICATION

A useful number of geometric transformations on arrays can be accomplished by modifying the fi, fj access tables, without any modification of the array itself. These transformations include rotation by multiples of 90 degrees, transposition, reflection, scaling, and translation. For instance, transposition is achieved by the exchange of tables between subscripts, reflection by reversing the content of the table, and rotation by a combination of transposition and reflection. Scale change and translation are accomplished by performing a linear mapping on the reference to the table entries. Is is improper to linearly map the actual values of the table entries.

VI IMPLEMENTATION

Good programming practice requires that algorithms should not have imbedded in them unnecessary information about the objects they manipulate. This notion, which is often called encapsulation, suggests that details of the storage of arrays should be of no concern to an algorithm that manipulates an array. Hence, the following functional forms of access are recommended:

```
value := get-element(array, i, j, ...)

put-element(array, i, j, ... , value)

boolean := is-element(array, i, j, ...)
```

Is-element returns true if the subscripts are within the bounds of the array.

Such arrays are constructed by the function:

```
array := new-array(type,size,ilb,iub,jlb,jub, ...)
```

This function allocates an array with subscript bounds determined by ilb, iub, ..., using an implementation-dependent default strategy to construct the tables. Type and size determine the data type and number of bits of the array elements.

An alternate form of array construction for user-specified transformation tables is:

```
array := new-array(type,size,i-table,j-table,...)
```

The bounds of the array are determined by the array bounds in the user-specified tables.

The bounds of an array can be accessed by:

```
get-bounds(array, size,ilb,iub,jlb,jub, ...)
```

where ilb, iub, ... are output parameters.

Input and output of such arrays are accomplished by:

```
array := get-array(file-name, mode)
```

Get-array constructs and returns a pointer to allow access to the array file on file-name. Mode specifies the permissible accessmode (read-only or read-write) for references to the array. It is assumed that no pages of array data are actually moved between virtual memory and the file until requested by element accesses to those pages. Therefore, array size does not significantly affect the time to access an element.

```
put-array(file-name, array, copy)
```

Put-array has the semantic effect of creating an array file named file-name, which corresponds to the data in the array. The underlying implementation details may vary, but in cases where the array is mapped to a temporary file, the parameter copy determines whether the data in the array is actually to the new file rather then renaming the temporary file.

VII DISCUSSION

The proposed additive decomposition scheme using tables to evaluate the functions fi and fj make implementation very efficient with conventional computers. When the array accesses are made in an order inappropriate to conventional array storage order, this scheme requires far less time than for conventional array-accessing schemes; otherwise, the accessing times are comparable (see Appendix C). This scheme also allows for the efficient access to images much larger than the virtual space of the machine.

Choices for these transformation functions that maximize the memory bit ratio depend on the characteristics of the computer system's memory hierarchy and the order of accesses to the array elements. It is conjectured that there are easily obtained models for system and algorithm access characteristics, from which a pragmatically optimum choice can be made for the subscript transformation functions.

A number of useful geometric transformations, such as rotation, transposition, translation, and scaling, can be performed by modifying these tables and require no modifications to the storage of the array.

The semantics of a set of procedures for array access, array creation, and the association of arrays with file names were defined. For computer systems with insufficient virtual memory, such as the PDP-10, a software virtual-to-physical mapping scheme is used. Implementations for the VAX and PDP-10 series computers to access pixels of large images stored as two-dimensional arrays of n bits per element are presented in Appendix C. Programs written in conformance with the proposed set of image primitives will be machine and operating system in ependent, which will facilitate the interchange of image understanding programs.

Appendix A

IMPLEMENTATION SEMANTICS

This appendix contains a functional specification written in a mixture of ADA and English for a package of procedures for two-dimensional array access to arrays of a single, fixed type. Appendix B contains a more complete example of how a subroutine package might be implemented. This specification defines an implementation-independent set of procedures and semantics for access to large arrays.

```
TYPE subscript IS INTEGER;

TYPE bigsubscript IS LONG INTEGER;

TYPE x_type IS  -- the type of the array elements

TYPE table_record IS
 RECORD
   lb,ub : subscript;
        -- lower and upper bounds for table
  element: ARRAY (lb .. ub) OF bigsubscript;
 END RECORD;

TYPE table IS ACCESS table_record;

TYPE x_type_array_record IS
 RECORD itab     : table;-- i transformation table
        jtab     : table;-- j transformation table
        elementtype: CONSTANT x_type_id;
        elementsize: CONSTANT x_type'SIZE;
                -- number of bits per element
```

The remaining fields are implementation dependent. For a virtual memory implementation, a pointer to an array of x-type is defined. For a software-mapped implementation, a page table of pointers to arrays of x-type are defined.

```
END RECORD;

TYPE x_type_array IS ACCESS x_type_array_record;

FUNCTION getelement(    p   : IN x_type_array;
                        i,j : IN subscript)
         RETURN x;
```

```
PROCEDURE putelement(   p       : IN x_type_array;
                        i,j     : IN subscript;
                        val     : IN x);

FUNCTION iselement(     p  : IN x_type_array;
                        i,j: IN subscript)
        RETURN BOOLEAN IS
        RETURN
            i IN RANGE p.itab.lb .. p.itab.ub
        AND j IN RANGE p.jtab.lb .. p.jtab.ub;

FUNCTION newxarray(
        si,sj   : IN subscript;
                -- number of columns and rows
        itab, jtab : IN table := NULL)
    RETURN x_type_array;
```

This function allocates an x_array with dimensions si, sj. The optional transform tables allow the user to specify his own coordinate transform strategy; otherwise, a CREATETRANSFORM uses implementation-dependent default strategy to construct the tables. NEWXARRAY calls CHECKTRANSFORM to check the range values of user-defined itab and jtab transform tables to guarantee that all possible accesses using them will be within the bounds of the x_array.

```
type ACCESSMODE is (readonly, writeonly, readwrite);

FUNCTION xarrayinput(
        filename : STRING;
        mode: assessmode := readonly)
   RETURN x_type_array;
```

XARRAYINPUT constructs and returns an xarraypointer to allow access to the array file on filename. Mode specifies the permissible accessmode for references to the array. It is assumed that no pages of array data are actually moved between virtual memory and the file until requested by element accesses to those pages. Therefore, the size of the array does not significantly affect the time to access an element.

```
PROCEDURE xarrayoutput(p        : x_type_array;
                       filename : STRING;
                       copy     : BOOLEAN :=FALSE);
```

XARRAYOUTPUT has the semantic effect of creating an array file named filename which corresponds to the data in array p. The underlying implementation details may vary, but the following effects are intended:

If p resides in memory, then p is written into an ARRAY file.

If p resides on a temporary file and copy=false, then the temporary file is renamed to filename.

If p resides on a nontemporary file or copy=true, then an exact copy of p is created on filename.

Xarray File Representation

The precise file format is implementation dependent, but must contain sx, sy, xtypecode, xtab, ytab, and all of the pages of the array data. The following format is recommended:

A word is defined to 32 or 36 bits, depending on the host computer. Pgsiz is the length of a virtual memory page on the host computer. A pointer within file is relative to the start of the file. A pointer within subfile is relative to word 0 of the subfile.

```
subfile
word
number
------

0: unique identifier for this subfile format
1: pointer within file to another subfile
2: sx number of columns
3: sy number of rows
4: xtypecode - a unique code identifying the data type x
5: xtypesize - the number of bits per element in data type x
6: pointer within subfile to the start of xtab
7: pointer within subfile to the start of ytab
8: pointer within subfile to the first page of array data
```

Appendix B

ADA DEFINITIONS FOR TWO-DIMENSIONAL ACCESS TO LARGE ARRAYS

```
TYPE subscript IS INTEGER;

TYPE bigsubscript IS LONG INTEGER;

TYPE x_type IS  -- whatever type you want here for
                -- the array elements

TYPE table_record IS
 RECORD
   lb,ub : subscript;
        -- lower and upper bounds for table
  element: ARRAY (lb .. ub) OF bigsubscript;
 END RECORD;

TYPE table IS ACCESS table_record;


TYPE x_type_array_body IS ARRAY (bigsubscript) of x;

FOR x_type_array_body USE PACKING;
        -- to maximize storage efficiency

TYPE xarraybodyrecord IS
 RECORD size : bigsubscript;
        element : x_type_array_body(0 .. size);
 END RECORD;

TYPE xarraybodyrecordpointer IS
        ACCESS xarraybodyrecord;

xarraypagesize : CONSTANT 512; -- for example

TYPE xarraypage IS
 RECORD
  data : x_type_array_body (0 ... xarraypagesize-1);
 END RECORD;

TYPE xarraypagepointer IS ACCESS xarraypage;

TYPE x_type_array_record IS
 RECORD itab     : table;-- i transformation table
        jtab     : table;-- j transformation table
        elementtype: CONSTANT x_type_id;
        elementsize: CONSTANT x_type'SIZE;
                 -- number of bits per element
        virtual : BOOLEAN;
```

```
            -- TRUE=> virtual implementation
        CASE virtual OF
         WHEN TRUE =>
          data: xarraybodyrecordpointer;
                -- this a pointer to the data
         WHEN FALSE=>
          numpages: INTEGER;
          pagesize: CONSTANT INTEGER xarraypagesize;
                -- for use by storage manager
          extdata : filedescr;
                -- a record structure for the file
                -- holding the data
          pagemap : array (1 .. numpages) of
                    xarraypagepointer;
        END CASE;
 END RECORD;

TYPE x_type_array IS ACCESS xarrayrecord;

FUNCTION getelement(    p    : IN x_type_array;
                        i,j  : IN subscript)
         RETURN x IS
BEGIN
  elenum: bigsubscript;
  pagnum : INTEGER;
  elenum :=  p.itab.element(i) + p.jtab.element(j);
  CASE p.virtual OF
    WHEN TRUE => RETURN(p.data.element(elenum));
    WHEN FALSE=>
        pagnum := 1 + elenum / pagesize;
        elenum := elenum MOD pagesize;
        IF p.pagemap(pagnum) = NULL THEN
                pagefault(p,pagnum);
        END IF;
        RETURN(p.pagemap(pagnum).data(elenum));
  END CASE;
END;

PROCEDURE putelement(   p        : IN x_type_array;
                        i,j      : IN subscript;
                        val      : IN x) IS
BEGIN
  elenum: bigsubscript;
  pagnum : INTEGER;
  elenum :=  p.itab.element(i) + p.jtab.element(j);
  CASE p.virtual OF
    WHEN TRUE => RETURN(p.data.element(elenum));
    WHEN FALSE=>
        pagnum := 1 + elenum / pagesize;
        elenum := elenum MOD pagesize;
        IF p.pagemap(pagnum) = NULL THEN
                pagefault(p,pagnum);
```

```
        END IF;
        p.pagemap(pagnum).data(elenum) := val;
  END CASE;
END;
```

The procedure PAGEFAULT is not defined here, since it will normally be implementation dependent. The intent is that space will be found large enough to store one xarraypage, and a corresponding xarraypagepointer will be stored into p.pagemap(pagenumber). Usually this will require searching a global table of pages to find either a free page or the least-recently-used page, which will be "kicked-out."

Appendix C

MACHINE-SPECIFIC IMPLEMENTATIONS OF IMAGE ACCESS

We now consider implementations of this scheme for both the VAX and the PDP-10, with two different assumptions: whether the entire image will or will not fit into an acceptable fraction of the computer's virtual memory space.

The implementer is encouraged to expend maximum effort to make the two-element access functions GETELEMENT and PUTELEMENT as efficient as possible, either through machine-coded subroutines, in-line code generation from within the host programming language, or code generation by the NEWXARRAY generation procedure. If the functions are efficiently implemented, researchers developing new algorithms will not be tempted to resort to special representations or coding tricks just to get more efficiency, and the resulting programs will be clearer and more transportable to any sites that implement these semantics. A proper implementation of these access primitives should be as efficient as most host languages will produce for standard two-dimensional array access.

We will first consider a VAX virtual memory implementation for variable pixel sizes. A MAINSAIL language definition of pixel access using the VAX variable bit field manipulation primitives could be written as:

```
INTEGER PROCEDURE getpix(
        POINTER(image) p;
         integer x,y);
        RETURN(extract-field(
                p.bpp*(p.xtab[x]+p.ytab[y]),
                p.bpp, p.picbase));

PROCEDURE putpix(
        POINTER(pix) p;
        INTEGER x,y,v);
        insert-field(v,
                p.bpp*(p.xtab[x]+p.ytab[y]),
                p.bpp, p.picbase);
```

where extract-field and insert-field correspond to the VAX variable bit field instructions.

A pair of very efficient VAX assembly language code sequences to access pixels using the MAINSAIL record storage conventions can be written as follows. These sequences assume that the pointer to the picture record is in register RP and the image coordinates are in registers RX and RY, respectively.

```
GETPIX: ADDL3 @XTABO(RP)[RX],@YTABO(RP)[RY],RTMP
         ; computes pixel number in image
        MULL2 BPP(RP),RTMP
         ; computes bit offset
        EXTZV RTMP,BPP(RP),PICBASE(RP),R1
         ; extract the pixel into R1
```

Note that XTABO and YTABO are offsets into the IMAGE record, which contains the virtual origins of the XTAB and YTAB tables. That is, the longword at XTABO+X corresponds to XTAB[X]. Also note that only one ADDL3 instruction is needed to compute the pixel offset.

```
putpix: ADDL3 @XTABO(RP)[RX],@YTABO(RP)[RY],RTMP
         ; pixel number in image
        MULL2 BPP(RP),RTMP
         ; compute bit offset
        INSV RVAL,RTMP,BPP(RP),PICBASE(RP)
         ; insert the pixel value in RVAL
```

The instructions shown in these procedures require only 8 data references plus 18 bytes of program reference. The overhead of a CALLS-type procedure call is certain to be more costly than the pixel access itself. Consequently, we recommend advised that either in-line code be generated for pixel access, or that a JSB-type procedure call passing arguments in registers be used.

The MULL2 multiply instruction can be eliminated by multiplying the values stored in the xtab and ytab tables by bpp as follows:

```
xtab[x] := xtab[x] * bpp
ytab[y] := ytab[y] * bpp
```

This scheme has been tested and requires about 9 microseconds per access with the MULL3 instruction and about 7 microseconds per access otherwise. For accesses to byte, word, or longword pixels, the EXTZV instruction can be replaced with the appropriate MOV instruction. This reduces the pixel access time to about 4 microseconds.

DEC PDP-10 Series Software Paged Implementation

The PDP-10 implementation is defined in SAIL. An image record class is declared as follows:

```
class pix(
        INTEGER sx;             image width
        INTEGER sy;             image height
        INTEGER bpp;            bits per pixel
        INTEGER ARRAY xtab;     x mapping table
        INTEGER ARRAY ytab;     y mapping table
        INTEGER ARRAY ptrtab;   byte pointers TABLE
        INTEGER ARRAY pagtab;   the mapping table
        INTEGER pagsiz;         words per page
        INTEGER fileid;         disk file id
        INTEGER gtpix;          entry to code
        INTEGER ptpix;          entry to code

);
```

Each image has a page table in its image descriptor record that the following information encodes for each page:

```
pagtab[page] = 0     ; page is not in memory
             = vpage ; virtual address of page
```

Each image also has a table of bytepointers, one for each pixel in an image page. This table can be shared among all images with the same pixel size and is constructed as follows:

```
ptr := point(bpp,0,bpp-1) + 4 lsh 18;
```

This constructs a bytepointer to the first byte in the word indexed by register 4 with byte size bpp,

```
for i:=1 step 1 until pagsiz do
 begin ptrtab[i] := ptr; ibp(ptr);end;
```

where IBP increments the bytepointer to the next byte.

A procedure PAGEFAULT manages paging tables and transfers data between the virtual memory and the file system. It is assumed that the read and write access modes are enforced by the pagefault procedure. For brevity, PAGEFAULT is not defined in this document.

The optimal PDP-10 code sequence is obtained by generating a version of the getpix and putpix procedures for each image, using the following PDP-10 assembly language sequences:

```
; gtpix assumes the registers are setup as follows:
; ac1: x
; ac2: y
; ac5: pointer to pic record
; ac6: return address
; result is returned in ac1

GTPIX:  MOVE    4,XTABBASE(1)
        ADD     4,YTABBASE(2)    ; element number
        HLRZ    2,4              ; offset(see note 1)
        SKIPN   4,MAPTABBASE(4) ; get virtual addr
        PUSHJ   P,PAGEFAULT      ; page not there
        LDB     1,PTRTABBASE(2) ; get the pixel
                ; byte pointers index by AC4
        JRST    @6

; ptpix assumes the registers are setup as follows:
; ac1: x
; ac2: y
; ac3: value to store
; ac5: pointer to pic record
; ac6: return address
; result is returned in ac1

PTPIX:  MOVE    4,XTABBASE(1)
        ADD     4,YTABBASE(2)    ; element number
        HLRZ    2,4              ; offset (note 1)
        SKIPN   4,MAPTABBASE(4) ; get virtual addr
        PUSHJ   P,PAGEFAULT      ; page not there
        DPB     3,PTRTABBASE(2) ; get the pixel
                ; byte pointers index by AC4
```

```
        JRST    @6              ; return
```

Note 1: For images with a power of 2 pixels per page, the xtab, ytab tables can be constructed so that no divide is required to compute the page number and offset within page. This is accomplished by modifying the access tables as follows:

```
modified-tab[i]:= tab[i] DIV pagsiz       page
                 +tab[i] MOD pagsiz LSH 18;  offset
```

These optimized procedures are attached to the image record fields GTPIX and PTPIX, respectively, and are not to be called directly from SAIL, but instead from these procedures:

```
INTEGER PROCEDURE getpix(POINTER(pix) pic;
                         INTEGER X,Y);
START!CODE
        POP P,6; return address
        POP P,2; y
        POP P,1; x
        POP P,5; pic
        JRST @gtpixoffset(4)
END;

INTEGER PROCEDURE getpix(POINTER(pix) pic;
                         INTEGER x,y,val);
START!CODE
        POP P,6; return address
        POP P,3; val
        POP P,2; y
        POP P,1; x
        POP P,5; pic
        JRST @ptpixoffset(4)
END;
```

Both SAIL and MAINSAIL packages have been implemented for access to variable pixel size, software-mapped image files. The following loop determines the pixel access times:

```
FOR Y:=0 STEP 1 UNTIL 255 DO
 FOR X:=Y STEP 1 UNTIL 255 DO
        BEGIN   INTEGER V1,V2;
                GETPIX(PIC,X,Y,V1);
                GETPIX(PIC,Y,X,V2);
```

```
            PUTPIX(PIC,X,Y,V2);
            PUTPIX(PIC,Y,X,V1);
    END;
```

The total CPU time on a KL-10 for a 256 x 256 image of 8 bits per pixel was 1 second, or about 8 microseconds per access including the loop overhead. Note that the GETPIX and PUTPIX functions used here were SAIL in-line macros rather than procedure calls.

The following loop that accesses the elements of a normal two-dimensional SAIL array in strictly storage order required about 8 seconds to execute, or about 8 microseconds per access:

```
SAFE INTEGER ARRAY A[0:255,0:255];
FOR K_1 STEP 1 UNTIL 16 DO
FOR I_0 STEP 1 UNTIL 255 DO
 FOR J_0 STEP 1 UNTIL 255 DO
        RESULT := A[I,J];
```

We conclude from these examples that if the pixel access functions are carefully implemented, there is little or no cost in time over conventional array accesses.