THE DATA ARRAY, A TOOL TO INTERFACE THE USER TO A LARGE DATA BASE

Garth H. Foster

Electrical and Computer Engineering Department

Syracuse University, Syracuse, N.Y. 13210

January, 1974

Final Report under NASA Grant NGR-33-022-150

for

Goddard Space Flight Center

Greenbelt, Maryland 20771

ABSTRACT

This report considers some aspects of the processing of spacecraft data and advocates the use of the data array in a large address space as an intermediate form in data processing for a large scientific data base. Techniques for efficient indexing in data arrays are reviewed and related to the data array method for mapping an arbitrary structure onto linear address space is shown, and a compromise between the two forms is given. Finally, some of the impact of the data array on the user interface and the implementation are considered.

ACKNOWLEDGMENT

The author would like to thank J. Boroumand and S. Hersh who provided some of the programming to support this effort. To Mr. G. E. Hoernes of the International Business Machines Corporation goes credit for the term multi-linear files. His interest in this problem should lead to additional material in this area.

TABLE OF CONTENTS

TABLES

FIGURES

1.0 INTRODUCTION

This report considers some aspects of the problem of interfacing a community of scientists to a very large data base. The primary assumptions are that: (1) Little is to be done concerning the collection and size of the raw data base; (2) Minimal burden should be placed on altering the scheme of intermediate data processing requirements or projections; (3) The types of data bases to be considered are such that archival system technologies probably will be pushed to the state of the art to accommodate the data base; (4) Output processing and the users point of view of the data base(s) would be the focal point of the material considered here in although there are implications in the earlier phases of the process.

The material which follows is apportioned as follows: Section 2 considers a somewhat simplified view of the data processing task; and the homogeneous, n-dimensional array, where each coordinate index is considered orthogonal to the others, is offered as a model for such data. The cases of commutation, sub-commutation, and super commutation as a part of the array data model are considered and the role of arithmetic progressions in describing these activities is explored. In this section, as well as the material to follow, the notation of A Programming Language (APL) is used to present the results and the algorithms. The reasons for this are that: (1) APL deals naturally with arrays such as considered here and the indexing properties of APL are very general; (2) some of the programs to be considered here were first published in APL and it is a natural notation for presenting those and other algorithms

here; and (3) APL is suitable for describing mathematical constructs, modelling programming in many high level languages, and describing operations which can take place on the machine level.

A working knowledge of APL is thus assumed. While references [1] and [2] are the usual one cited, for the purposes considered here the beginner should probably start with [3] or [4] and use [2] as a reference.

Section 3 considers the n-dimensional array and reviews the requirements for mapping this regular structure onto linear address space. What is known about efficient implementation of indexing in APL-like arrays is presented next and related to the commutation results and data array indexing given earlier. The structure such as found in pL/1 or COBOL is shown to be a generalization of the array and it is considered next. A function for mapping such structures onto linear address space is given in APL. This algorithm calculates the addressing parameters for an arbitrary structure. By restricting the form of a structure we arrive at a generalization of the array which has utility in the problem under study having addressing parameters that are less compact then those used in an APL array, but more compact than those used in mapping an arbitrary structure.

Section 4 deals briefly with the user/data base interface. Choices here depend on what is to be optimized and the constraints of the problem, profile of data access requests, machine architecture of the output and/or retrieval processors and other factors which can not be adequately dealt with in this report affects such considerations. Accordingly, trends are noted, and possible choices are outlined.

The final section deals with some observations regarding machine structure since the scope of this effort precludes a more complete design effort, the salient points of view of this effort are reflected into machine characteristics which would tend to make an approach such as suggested here more feasible.

2.0 MODELS FOR TELEMETRY DATA PROCESSING

2.1 Background

The nature of the data processing requirements and workload at Goddard Space Flight Center strongly impacts the interface between the user and the data base(s) of interest to him. First, the sheer volume of data produced by each spacecraft, not counting the total volume of data from all spacecraft, places the problem on or near the leading edge of technology. The second factor comes from the fact that the data processing task associated with telemetry data processing, the step prior to solving the user data base interface, is itself a computationally robust problem.

To seek a model into which we may cast the data base problem we assume that each spacecraft has a number of digital sensors the values of which are measured in a position-time sequence designed for that spacecraft's payload. A number of data words are transmitted, together with patterns to assist in synchronization and recovery of information in the face of noise, as a frame of data.

The data is transmitted to a number of ground stations which forward it via the NASA Communications Systems (NASCOM) to Goddard. Table 1

summarizes how we choose to categorize computational activity beyond this point. It is likely that this table neither reflects all steps of telemetry processing for all present day satellites nor does it quite have divisions of activity phases which reflect trends at GFSC. The groupings chosen were primarily taken from descriptions given in reference [5, 6] and to some extent [7]. These steps do not reflect the activity that would be considered data reduction, interpretation and/or analysis such as might be described in [8] and used for quick-look or more extensive analysis.

Past practice has found that step 4 of Table 1 was essentially the writing of a master tape. Step 5 was the simultaneous decommutation of all experiments with a tape being prepared for each experiment (or perhaps for each experimentor if a team were working on a single task). Of course the simultaneity was subject to the requirement that the number of tapes to be made not exceed the number of drives available.

Current trends are aimed at using improved technology to attack a number of problems. Some of these are timelines of processing, volume of data and output processing on demand. To speed up availability of telemetry data it is understood that the Station Data Acquisition and Control (STADAC) system is transferring some processing activity, such as the production of digital tapes, from the computer to the data acquisition sites. Moving in the direction of near real-time processing is dictated not only by a desire to get data to experimenters in a more timely manner but also because some spacecraft, such as the recently launched Atmospheric Explorer series, require rapid response to ensure

PHASE	ACTIVITIES INCLUDED
1. Raw Data Processing	a) Reverse playback data
	b) Perform convolutional decoding
	c) Check frame and subcommutator synchronization
	d) Flag errors (and perform corrections where possible)
	e) Make checks on spacecraft housekeeping, time, etc., and flag errors
	f) Determine format changes
2. Initial Refinement	a) GMT time determination
	b) Time smoothing
3. Correlation and Editing	a) Attitude and orbital correlation
	b) Edit processing
4. Archival Storage	a) Transfer of data between appropriate media
	b) Create directories indicating location level, and media of data
5. Decommutation and Output Processing	a) Decommute experimental data
	b) Append orbital and attitude data
	c) Provide format changes
	d) Format output
	e) Prepare hard copy notification and/or documentation.

Table 1 - Processing Functions, Summarized

the well being of the experiments, as well as the spacecraft.

Another trend apprears to be rather than each experiment being unique to an individual or location, a number of people in diverse locations will be interested in the data from a single experiment. In addition not all will be interested in all of the data and output processing on demand is one approach to such requirements. Increased storage capacity is needed to be able to handle and store the increased volume of data which longer spacecraft lifetimes and increased numbers of sensors imply.

Phases 1 and 2 are often combined but for the purpose of this discussion we have indicated a separation as shown because the kind of processing requirements needed in phase 1 has recently been shown by Broglio [8] to be easily fulfilled by a specialized processor designed to perform such pre-processing tasks. The calculations in the second phase as listed in Table 1 seems to be computationally more burdensome than the activities indicated by Broglio. However, if these activities can be added to the tasks of such a specialized processor, there appears no reason why this should not be done.

The next activity (3) is described as varying from one spacecraft to the next, and it is identified as an individual task for that reason. The assumption is made that the correlation tables for orbit and/or attitude data are generally not merged with the experimental data for which position is to be supplied. The archival storage function of TELOPS [6, 7] is essential to that system and marks what seems to be a significant departure in the processing of space data. It is to the use

of archival storage and output processing that this report speaks. Decommutation has, as indicated earlier, generally been done once as scheduled. Demand processing, particularly where the number of users of the data is not vanishingly small, implies that requests for decommutation may not arrive fortuitously, causing additional processing burden due to repeating demand output.

While we assume that an economical input processor: (1) places data in a uniform form regarding the position of the least significant bit, (2) orders data chronologically, and (3) aligns data (word, half word, byte or bit) boundary as may be required by successive processors.

Intermediate processing will be done by conventional systems and the question may then be asked: "Are there any reasonably general models which will be useful in the archive and/or output phases of processing?"

2.2 Data Structures and Arrays

The physical nature of storage media has always placed constraints on the way we treat logical-to-physical mappings in the storage of data and on the way in which we perceive the organization of the data in the beginning.

The one dimensional nature of tapes has strong implication as to the kinds of data which may be efficiently mapped to that medium. Items having a strong sequentiality are most easily related in logical-to-physical mappings for tapes. Situations where there is a hierarchy in the structure can be mapped to tapes but with little liklihood of either easy expansion of the data base or efficient random retrieval of

information.

At the same time, the low storage cost per bit has made tape attractive as a storage medium for large data bases. Tape has served as a reasonably good compromise for the storage of data such as that of interest to GSFC because the low storage cost per bit and the high volume of data are compatible; and since the number of transmitters operating in parallel is reasonably small, the data flow is primarily sequential in nature rather than parallel. The multiplicity in transmitters may be handled by an n-fold parallelism of the discussion to follow or by considering the net result to be a single source operating at an increase bandwidth.

Accordingly we consider the spacecraft to station contact to be a cast in the format of a single serial transmission directly correlated with time. Successive station contacts may be brought to universal time and ordered chronologically.

This for the moment neglects the case where the spacecraft may be fading out of contact with one station while coming into contact with another and duplicated at both stations. Problems which may exist with playback and real-time data reception for related reasons are also excluded.

The tacit assumption is made that the use of disk or traditional rotational memories has been somewhat more limited in the use because of the data volume and storage costs except for cases where the volume is kept manageable by limiting the period of retention. Quick look analysis or selected positions of the orbit would be examples.

As the archival storage devices discussed in references [6, 7] become available, the intent to have data stored on-line infers that different accessing characteristics will play a role in system efficiency. The net effect of rotational or bulk memories which behave more like disks than tape is to come closer to approximating a memory which has random access.

Of course main storage has linear addressing but the constant time to access the next, arbitrarily chosen, memory cell leads us to conceptional organizations of the data which are other than linear. For example, several indirections in addressing directory search or chaining through a tree structure are relatively small compared to tape or disk access times.

2.3 An Array Model

Consider a frame of *DATA* which, for the nonce, is idealized in the following way: A frame is P bits long and there are N experiments each of which require M bits of data to encode that sensor's output. Clearly $P \leftarrow\rightarrow M \times N$, where we use the notation $L \leftarrow\rightarrow R$ to denote that L" is equivalent to "R.

If we were to consider the data not as a vector of bits but as a matrix, then *FRAME* $\leftarrow$ $(N,M)\rho$*DATASTREAM* and for $I \epsilon \iota N$

FRAME$[I;]$ $\leftarrow\rightarrow$ the Ith sensor value (M bits). By the same token, *FRAME*$[;K]$ may be considered to be the Kth bit of <u>all</u> experiments. It is not clear why we would be interested in such a construct particularly since the bits will be in different computer words in main store and therefore will be extremely costly to extract. If conditions are such

that M, the number of bits is, say 8 so that each sensor is a byte and if a computer word is 4 bytes then the natural arrangement of data is: *FRAME*4← ((*N* ÷ 4), 4 × *N*)ρ*DATASTREAM*. Now if *K* is of the form *K* ← *BYTEBOUNDARY* × ⍳8 (in index Origin 0), *FRAME*4[*J* ; *K*] denotes the *I*th experiment if *J* ← → ⌊*I* ÷ 4 and *BYTEBOUNDARY* ← → 8 × 4|*I* (also in Origin 0). Where *K* gives the indices of the bits within the word. Here the matrix model has conceptual value in describing packed data when the packing aligns the data in a fashion which is related to the addressing structure of the host computer.

Returning to the first approach, suppose we have *T* frames and data then *PACKET* ← (*T*, *N*, *M*)ρ*DATASTREAM* is a three-dimensional array where *PACKET*[*I* ; ;] is the *I*th frame and *PACKET*[;*J*;] is the *J*th experiment over all *T* frames. If there are *S* sources, then we may assume without loss of generality that the frames are interleaved such that *STATIONCONTACT* ← (*T*, *S*, *N*, *M*)ρ*DATASTREAM* is a conceptual model for addressing somewhat orthogonal but related quantities.

STATIONCONTACT[*I* ; ; ;] is the collection of all *I*th frames from all sources.

STATIONCONTACT[; *J* ; ;] is the collection of all frames from the *J*th source, and

STATIONCONTACT[; ; *K* ;] is the *K*th experiment in all frames from all sources. In this last case we have made the (generally unfounded) assumptions that each transmitter on board the spacecraft provides data from the same number of experiments on each frame and further that the *K*th experiment from source A is related to that from source B.

Still the conceptual appeal of the array, even with the present restrictions, lead us to inquire whether there are: (a) other descriptive uses to which the homogeneous parallelepiped array may be applied, (b) mappings which take us from the multi-dimensioned array to the linear mappings which are natural to main store, and (c) adjustments which allow us to deal with the restrictions in applying arrays to the problem at hand.

We offer a brief look at the problem of commutation relative to the first issue above and we defer the other two to later sections of this report.

2.3.1 Commutation

Suppose that we assume that each data value fits into a computer word so that the three dimensional array previously encountered may be thought of as a matrix with the first coordinate direction denoting the frame number and the second providing the experiment index. Thus suppose that *DATA*[;*K*] is actually a commutation of *S* experiments. For a scalar, *K*, *DATA*[;*K*] is the vector representing the commutated data; and if we are interested in the *J*th slot on the commutator then we want the values given by the selection expression $(J = S|\iota\rho DATA[;K])/DATA[;K]$ where of course we require $J\epsilon\iota S$. Now if we want to use the indexing of an array as our model we need to achieve the form *DATA*[*VECTOR* ; *K*].

Since the compression above selects the *J*th element in the sequence and then values *S* apart in the vector *DATA*[;*K*] we expect *VECTOR* to have the form $START + SPACING \times \iota SIZE$ in Origin 0, and this is an arithmetic progression. Thus, $VECTOR \leftarrow\rightarrow J + S \times \iota SIZE$ where

$SIZE \leftarrow\rightarrow (J \leq S|\rho DATA[;K])+ \lfloor(\rho DATA[;K])\div S$. The expression for $SIZE$ simplifies if we have enough frames ($\rho DATA[;K]$) such that the commutator size S divides evenly the number of frames so that $0 \leftarrow\rightarrow S|\rho DATA[;K]$ then $SIZE \leftarrow\rightarrow (\rho DATA[;K]) \div S$.

The indexing expression $DATA[J + S \times \iota(\rho DATA[;K])\div S ; K]$ selects the commutated values.

As an aside we should note that if we seek rigor we sould formally establish that the forms

$$U/V \leftarrow\rightarrow U/V[\iota\rho V]$$

$$\leftarrow\rightarrow V[U/\iota\rho V]$$

$$\leftarrow\rightarrow V[A + B \times \iota C]$$

are equivalent for proper choices of A, B, C relative to U and V. Examples of formal proofs in APL can be found in [10]. The first equivalence is trival in that the vector generated by its own size. The second transformation follows from Lemma L6.2 of Abrams [10] (p. 42). The final form follows from the fact that U has ones evenly spaced because $S|\iota N \leftarrow\rightarrow N\rho\iota S$. Such a spacing of ones implies indices having an arithmetic progression.

Next, unless the host computer has some unusual addressing capabilities, the vector index expression of the form $D[J + S \times \iota(\rho D) \div S]$ is not a good model. We delay in providing an alternate formulation to point out that commutation may be expressed as indexing an array with a suitably chosen arithmetic progression vector.

2.3.2 Supercommutation

Supercommutation takes place when more that one slot on a commutator ring is used to capture the data for a single experiment, thus allowing the sampling rate to be higher. These positions in the ring are denoted by the vector *LIST* where *LIST* $\in \iota S$ and the positions of the slots are in sequence such that $I \leq J$ implies *LIST*[*I*] $\leq$ *LIST* [*J*]. Then the supercommutation may be expressed by

DATA[,(*S* × ι(ρ*DATA*[;*K*])÷ *S*)∘.+*LIST*;*K*]

In general, LIST will itself be an arithmetic progression as this will provide an even spacing around the ring. In such a case the expression for the first coordinate index is of the form (still in Origin 0): ,(*S* × ι(ρ*DATA*[;*K*]) ÷ *S*)∘. + *A* + *B* × ι*C* with *A*, *B*, *C* being positive integers which will give the arithmetic progression in *LIST*. The net effect of such an expression can be obtained by two nested loops such as:

```
INDEX ← ι0
I ← 0
OUT:  J ← A
IN:   INDEX ← INDEX, I + J
 → IN IF (A+B×C) > J←J+B
 →OUT IF (ρDATA[;K])>I←I+S
```

If *LIST* has only one element in it then *C* ← → 1 and ι1 is zero in Origin 0 comsequently the test at 1+ *IN* fails and falls through every time. The effect is only one loop starting at *A* and moving forward in increments of *S*. This then reduces to the earlier discussion on commutation.

2.3.3 Subcommutation

Subcommutation is a commutation within a commutation. To fit this to the model we need to study $(H = Q|\iota P)/VEC[A + B \times \iota C]$ where the expression involving H, Q and P denote the subcommutation. By previous arguments this can be considered as $(VEC[A+B\times\iota C])\ [T+U\times\iota V]$. Now by Lemma L1 of Abrams [10] (p. 43), this is $VEC[(A+B\times\iota C)[T\times U\times\iota V]]$. The question of $(A+B\times\iota C][T+U\times\iota V] \leftarrow ? \rightarrow K+L\times\iota M$ for the choices

$M \leftarrow\rightarrow V$

$L \leftarrow\rightarrow B \times U$

$K \leftarrow\rightarrow A + B \times T$

is answered in the affirmative. Thus, we conclude that subcommutation fits the same model.

Reducing both of these cases to vector indexing implies that the model chosen also handles, by composition of mappings on the indices, a subcommutation of a supercommutation of the data. A direct calculation of the final index set is less than straight forward because of the nested loops which derives from the outer product used to represent the supercommutation. The indexed array model may still be viewed another way and that is if $DATA[;K]$ for scalar K gives a single experiment and supercommutation then becomes $DATA[;SLOTS]$ with the vector $SLOTS$ being suitably chose to represent the encoding used. A reshape allows subcommutation indexing, giving the form $((M,N)\rho DATA[;SLOTS])[A+B\times\iota C;K]$.

This foregoing convinces us that the array model is viable in the case of uniform data.

2.4 Multi-Linear Files

In the above we have been considering the array, a parallelepiped arrangement of data, as a suitable vehicle for conceptualizing spacecraft data. It appears that considering a data base to have a number of orthogonal indices, where in any particular coordinate direction there is the concept of a successor and predecessor of a data value, is a useful approach. The structures of hierarchies, trees, linked lists and so forth are replaced by the regularized structure of the multi-dimentional array. The concept of the multi-linear files is being studied by G. E. Hoernes at this loaction and his work will be published elsewhere. While Mr. Hoernes is concerned with problems of searching along arbitrary directions and of defining bounded sub arrays of the original, or host, array, his early work indicates that the conceptualization which we have advocated here for spacecraft data is useful in relation to other data bases.

The multi-linear data base is primarily one of stored information from which data is retrieved for use in calculations elsewhere as opposed to data which is accessed calculated and updated, although such an approach is also possible. The data base grows in one of the coordinate directions only and that direction is usually strongly correlated with time. The other dimensions are generally fixed in number and the size of each does not change except, rarely over the history of the data base.

It is beyond the scope of this work to cover more than a few aspects of how Multi-Linear Files may relate to data base organization. Searching the values within some subarray to find the indices which correspond

to these values meeting prespecified criteria in what amounts to an associated research is interesting but does not relate directly to the problem at hand.

Data arrays which have holes or sparse volumes also do not apply because the model postulated earlier is reasonably dense. Techniques for handling duplicate values are more relevant and we shall allude to this problem is a subsequent section.

Finally, there are a number of logical to physical mappings which can be applied to multi-linear files. We now turn our attention to techniques of such mappings for the array indexed model introduced in this section.

3.0 STORAGE MAPPING FUNCTIONS

To make the array model useful we need to study the way in which an element of the array is mapped onto physical storage. In the following discussion we assume that all available storage is addressible and some unit of storage, such as a word, is obtained on an access. This means that the data has an address space in the range $\iota 2*M$. This implies a virtual storage or at least a storage management system which performs roll-in/roll-out on segments of data which are large enough to make the overhead of the software system acceptable. The second requirement for such a storage manager is that the statistics of use are such that most of the time we will remain within the segment rolled into main storage so that excessive thrashing does not occur. To begin with we review array storage and indexing in APL and then examine other storage strategies useful for the problem at hand.

3.1 Arrays and Indexing in APL

Arrays in APL are: rectangular, in the sense that coordinate directions are orthogonal, dense, in the sense that space is allocated for a value at every index and either the entire array has values or it is empty (having a dimension such that $0 \leftarrow \rightarrow \times/\rho ARRAY$), homogeneous, in that the packing of items in the array and the space reserved for each is uniform. The last property is strongly one of implementation convenience and not a requirement of the language.

Since each array is dynamically alterable in terms of its size as well as the values that it holds, this structure information must be retained together with the array's values. Each array is stored in lexiographic or ravel order (row major order for matrices). Thus arrays generally appear in storage as:

HEADER,($\rho\rho$*ARRAY*), (ρ*ARRAY*),,*ARRAY*

HEADER contains packed information such as back pointers to the symbol table, information as to type and whether the array is currently unused and hence garbage, and a length count of the entire array. The value of $\rho\rho$*ARRAY* is included to give the size of ρ*ARRAY*. The values contained in ,*ARRAY* may be different than the storage requirements of *HEADER* and/or $\rho\rho$*ARRAY* and ρ*ARRAY* and so the packed structure is of record type however, the *HEADER* is of fixed size and the rest of the structure information is calculated from the rank of the array.

To address the value of *ARRAY*[*I* ; *J* ; *K*] (for scalar I ; J ; K) let us consider that 30 20 10 $\leftarrow \rightarrow \rho$*ARRAY*. We then note that there are *I* -1 planes before the one of concern (in Origin 1), *I* - 1 rows and K

elements including the addressee. Each plane has 10 × 20 ← → 200 elements in it, and each row has 20 elements in it. Thus, *ADDRESS* ← +/200 20 1 × (*I* - 1),(*J* - 1),*K* is the value needed for (,*ARRAY*)[*ADDRESS*]. This may be put in a more uniform form by rewriting this as *ADDRESS* ← 1 + +/*W* × (*I*,*J*,*K*) - 1 where *W* is 200 20 1. Noting that subtracting by one changes indices to Origin 0 and adding one returns to Origin 1, we may rewrite the expression as: *ADDRESS* ← *IORG* ++/*W* × (*I*,*J*,*K*) - *IORG* where *IORG* denotes the index origin.

Finally noting that the calculation of *W* is precisely the weighting vector used in calculating the base value in the mixed radix system (ρ*ARRAY*), we may note that the translation of the array index to an index of the ravel of the array is

(,*ARRAY*)[*IORG* + (ρ*ARRAY*) ⊥ *INDEX* - *IORG*]

INDEX ← *I*1, *I*2,... *IK*

with the *I*i denoting the scalar index values of each of the *K* coordinates. This is just a polynomial evaluating the indices in a mixed radix number system.

The problem of efficient indexing in APL has addressed by Hassitt and Lyon [11], and before we review their results it is pertinent to note why the structure of APL provides a problem of efficiency. In general, array *A*, of rank *K* (i.e. ρρ*A* ← → *K*), is indexed by an index list with the *K* index expressions being separated by *K* - 1 semi-colons. Let [denote *SEPARATOR*(0). The *K* semi-colons denote *SEPARATOR*(*I*) and] denotes *SEPARATOR*(*K*+1) then for *K*∈ιρρ*A*(*ORIGIN* 0),the expression

between *SEPARATOR(K)* and *SEPARATOR(K + 1)* may be:

a) Missing - In which case take the expression to be

$E \leftarrow\rightarrow \iota(\rho A)[K + IORG]$

b) Some APL expression E the values of which are in the domain of that subscript $1 \leftarrow\rightarrow \wedge/,E\in\iota(\rho A)[K + IORG]$.

Noting that the shape of resulting array is the catenation of the shape of the K expressions and the rank of the result is the sum of the ranks of the K expressions, suppose E_k, the expression between the and $K + 1$ separator, is of rank:

0 - This dimension's index tends to reduce the rank of the result and the effect of this dimension is constant over the rest of the array and it may be calculated once and removed from the remaining calculations

1 - This subscript neither decreases nor increases the rank of the result. The special cases of an arithmetic progression (AP) vector here or missing expression (equivalent to an AP vector) are worth noting because only 3 parameters are needed: START, STEP size and LENGTH; otherwise we step through the values.

>1 - This subscript tends to increase the rank of the result and usually the fact that the ravel of the values form an AP vector either does not hold or such information is lost by the time evaluation is carried out.

As shown above the evaluation of an array, A, subscripted in each of the K dimensions with a scalar requires $K + 1$ multiplications in the weighting process. If we let E_i represent the index expression in the ith coordinate and letting S denote a vector which relates to the original array A in the following way:

$\rho S \leftarrow\rightarrow K \leftarrow\rightarrow .\ \rho\rho A$

$S[I] \leftarrow\rightarrow \rho,E_i.$

We have $S[I]$ as a number of elements used in the ith index direction, independent of the size and rank of the value of the ith subscript expression. Then one would believe that $\times/S$ evaluations of a scalar index are required implying $(K - 1) \times \times/S$ multiplications. Using the technique of Hassitt and Lyon [11] this may be reduced to at most +/S multiplications, and we summarize their technique next.

3.2.1 Hassitt and Lyon's Approach to Indexing

Figures 1 through 4 give modifications to the results published in [11]. This was done for two reasons:

(1) The routines in the reference cited worked with a character string representation of the indexing operation and data structures of that string (as would be done internal to APL); what we have done is to write two new functions *INDEX* and Δ so the indexing process on array, *ARRAY* could be modeled by *ARRAY INDEX* (E_0) Δ (E_1) Δ ... Δ (E_k) where E_i is the index expression, (variable, or constant) in the ith coordinate position. This allows the execution to be traced. The functions INDA and POP of [7] were modified to account for this change.

and (2)

Three errors were discovered in *INDC* as given in reference [11]. In line [23] the >0 should be replaced by ≥0 to account for the fact that the functions run in 0 origin indexing and 0 is a valid subscript. Secondly, in the text as published, the variable U is not initialized in *INDC*[40] when using the call *INDC* 1. Thus,

INDC[40]*ENDS*: $S\leftarrow 256|L[U\div 4]$ should be corrected to read:

```
      ∇INDEX[□]∇
    ∇ Z←A INDEX R;P;J;K;M;D
[1]   →(1≠ρρA)/L01
[2]   Z←A[R]
[3]   →0
[4]  L01:P←(ρρA)-1
[5]   J←1
[6]   B←R[0]↑R
[7]   K←R[0]+3
[8]  LOOP:M←K+R[K]
[9]   D←K↓M↑R
[10]  B←B,D
[11]  J←J+1
[12]  →((J=P),(J>P))/L2,L3
[13]  →LOOP,K←M+3
[14] L2:→LOOP,K←M
[15] L3:IB←(J←1)↑B
[16] L5:IB←IB,(+/IB)↓(1++/IB)↑B
[17]  →(P≥J←J+1)/L5
[18] L4:INDA
[19]  INDB
[20]  Z←EVAL
    ∇
      ∇Δ[□]∇
    ∇ R←Z Δ Y;FY;FZ
[1]   FY←(ρρY),(ρY),Y
[2]   FZ←(ρρZ),(ρZ),Z
[3]   R←(1+ρFZ),FZ,(1+ρFY),FY
    ∇
```

```
      ∇BIT[□]∇
    ∇ Z←BIT K
[1]   ⍝ GET BIT K OF S
[2]   ⍝ NOTE THAT 0 ORIGIN IS USED
[3]   Z←(,(8ρ2)⊤S)[K]
    ∇
      ∇CHECK[□]∇
    ∇ Z←CHECK A
[1]   →(∨/0>,Z←A)/ERROR
[2]   →(~∨/⌊≤,A)/0
[3]  ERROR:'INDEX ERROR'
[4]   →
    ∇
      ∇ERRORIF[□]∇
    ∇ A ERRORIF C
[1]   →0 IF~∨/,C
[2]   A,' ERROR'
[3]   →
    ∇
      ∇IF[□]∇
    ∇ Z←A IF B
[1]   Z←B/A
    ∇
      ∇IORG[□]∇
    ∇ Z←IORG
[1]   Z←⎕IO
    ∇
```

```
      A
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
18 19 20 21 22 23

24 25 26 27 28 29
30 31 32 33 34 35
36 37 38 39 40 41
42 43 44 45 46 47

      A[0;2;1+2×⍳3]
13 15 17

      A INDEX ((0)Δ(2)Δ(1+2×⍳3))
13 15 17
```

```
      A[ ; ;2 3]
 2  3
 8  9
14 15
20 21

26 27
32 33
38 39
44 45

      A INDEX ((⍳0)Δ(⍳0)Δ(2 3))
 2  3
 8  9
14 15
20 21

26 27
32 33
38 39
44 45
```

Figure 1 Index and Ancillary Functions

INDC[40] *ENDS*: $S \leftarrow 256|L[U \leftarrow \lfloor L[1] \div 4]$. Finally, the values of *COUNT*, *STEP*, and *Z* were not changed in executing *INDC*[40] through *INDC*[50]. Line [49] is in error and should be corrected from:

INDC[49] *DOWN*1: *STEP* ← *L*[*U* + 2] ← *L*[*U* + 5] to:

INDC[49] *DOWN*1: *STEP* ← *L*[*U* + 2] ← *STEP* + 1

In the repeated use of Δ, a block $(2+(\rho\rho E_i)+\times/\rho E_i),(\rho\rho E_i),(\rho E_i),,E_i$ is built up for each index expression E_i. Thus, Δ acts as a semicolon in writing the expression and the parentheses are required by APL's order of evaluation. Missing subscripts require an explicit (⍳0) to be inserted. This compound right argument is decoded and moved to *INDA* (modified) via *POP* (also modified).

In *INDA* a vector *L* is constructed (in effect) by the catenation of a number of other vectors $L \leftarrow\rightarrow C, B0, B1, \ldots, BK, V, S, T$.

Think of *C*, *B*0, *B*1,..., *BK* to be the ravel of a matrix *CB* of *K* + 2 rows and 6 columns; then (*CB*[0;] ← → 8, (1 6 × *K* + 1), 3ρ0 and for $1 \leq I$ and $I \leq K + 1$, *BI* ← → *CB*[*I*;] is of the form:

$0,E_i$, 0 0 0 0	if E_i is scalar
$1,E_i$[0], 0 0, (ρE_i), -/E_i[1 0]	if E_i is an APL vector
5 0 0 0, *OS*,1	if E_i is missing
2 0 0 0, (ρE_i), *OV*	if E_i is a vector or array

The meaning of *OS* and *OV* will be noted in a moment.

CB[; 2 3] will be used to hold ρ*A* and the polynomial weights repsectively. *V* contains, in sequence, the catenation of the ravels of all blocks (index expressions) which are not scalar, AP vector, or missing.

S is the shape of the result; $T \leftarrow \rightarrow \rho S$ is the rank of the result. In the case of a missing subscript, OS points to the location of L (actually in the S portion) where that size will be found. This will be filled in later when the missing subscript is converted to an AP vector. When E_i is a vector (not AP) or array, OV (i.e. $CB[I;5]$ points to the place in L (in the V portion) where the values from E_i begin.

In *INDC* a number of intermediate calculations and optimizations take place. In the following the calculations values are placed in L but we will describe them in terms of CB.

First the shape of A and the weights are filled in, (remembering that $\rho\rho A \leftarrow \rightarrow K + 1$)

$CB[1+\iota K+1;2] \leftarrow \rho A$ then the weights are derived:

$CB[2+K;3] \leftarrow 1$

MORE: $J \leftarrow K + 1$

$CB[J;3] \leftarrow \times /CB[1+J;2\ 3]$

$\leftarrow MORE\ IF\ 1 \leq J \leftarrow J-1$

Calculations then proceed to: check if each subscript is in range, multiply scalar values ($CB[I;1]$ when $CB[I;0] = 0$) by the weight $CB[I;3]$ and add to $CB[0;0]$ (the fixed part of the address). Missing subscripts are then made to look like AP vectors. AP vectors are checked for range and multiplied by the appropriate weight $CB[I;1\ 5] \leftarrow CB[I;3] \times CB[I;1\ 5]$. Each of the appropriate values in V is multiplied by its appropriate weight.

Finally adjacent AP vectors are examined to see if only a single loop may be used rather than running counters for nested loops. This

gives optimization to indices of the form *A*[*K*; ; ;].

Suppose the ρ*A* ← → 300 100 20 and the expression in *A*[3 ; ;], then if we display *L AS*

CB

V

S

T

we have at the end of *INDA*

8	3	18	0	0	0	← → *C*
0	3	0	0	0	0	← → *B*0
5	0	0	0	24	1	← → *B*1
5	0	0	0	25	1	← → *B*2
0	0					← → *S*
2						← → *T*

V does not appear since there are no indices which are not AP vectors or which have general arrays. The 24 and 25 in *CB*[2 3 ; 4] point to the location of the two zeros in *S*.

After moving the dimensions and weights in, the structure would look like:

8	3	18	0	0	0	← → *C*
0	3	300	2000	0	0	← → *B*0
5	0	100	20	24	1	← → *B*1
5	0	20	1	25	1	← → *B*2
0	0					← → *S*
2						← → *T*

```
      ∇INDA[□]∇
    ∇ INDA;LL;ARANK;SRANK;BB;BS;BV;N
[1]   LL←7+ARANK←SRANK←0
[2]   J←0
[3]  L1:ARANK←ARANK+1
[4]   →(T0,T1,T2, 0 0 ,T5)[POP]
[5]  T5:→T1
[6]  T2:LL←LL+ρ,X
[7]   →T0,SRANK←SRANK+ρρX
[8]  T1:SRANK←SRANK+1
[9]  T0:LL←LL+6
[10]  →((ρρA)>J←J+1)/L1
[11]  L←(LL+SRANK)ρ0
[12]  L[0 1 2]←8,ARANK,6×ARANK
[13]  L[¯1+ρL]←SRANK
[14]  BB←6
[15]  BV←L[2]+6
[16]  BS←(ρL)+¯1-SRANK
[17]  J←0
[18] L4:→(TT0,TT1,TT2, 0 0 ,TT5)[POP]
[19] TT5:L[BB+ι6]← 5 0 0 0 ,BS,1
[20] L2:→L3,BS←BS+1
[21] TT2:'DOMAIN' ERRORIF∨/,X≠⌊X
[22]  L[BV+ιN←ρ,X]←,X
[23]  L[BS+ιρρX]←ρX
[24]  BS←BS+ρρX
[25]  L[BB+ι6]← 2 0 0 0 ,N,BV
[26]  →L3,BV←BV+N
[27] TT1:L[BS]←ρX
[28]  →L2,L[BB+ι6]←1,X[0], 0 0 ,(ρX),X[1]-X[0]
[29] TT0:'DOMAIN' ERRORIF X≠⌊X
[30]  L[BB+ι6]←0,X, 0 0 0 0
[31] L3:BB←BB+6
[32]  →((ρρA)>J←J+1)/L4
    ∇

      ∇POP[□]∇
    ∇ PP←POP;H;I;LI;RI
[1]   →(J≠0)/L6
[2]   I←(IB[J])↑B
[3]   →L7,NB←IB[J]
[4]  L6:I←NB↓(NB+IB[J])↑B
[5]   NB←NB+IB[J]
[6]  L7:→((I[1]=0),(I[1]=1),(I[1]≥2))/LL1,LL2,LL3
[7]  LL1:X←2↓I
[8]   →PP←0
[9]  LL2:→(I[2]≠0)/LL4
[10]  X←ι,(ρA)[J]+1
[11]  →0,PP←5
[12] LL4:X←3↓I
[13]  →((ρX)<3)/LL5
[14]  →(∧/(⌈/T)=T←(1↓X)-¯1↓X)/LL6
[15] LL5:→0,PP←2
[16] LL6:→0,PP←1
[17] LL3:RI←2↓(2+I[1])↑I
[18]  LI←(2+I[1])↓I
[19]  X←RIρLI
[20]  →0,PP←2
    ∇
```

Figure 2 INDA and POP

Next the blocks are processed checking for bounds; missing subscripts are converted to AP vectors with the array sizes filled in. The scalar value is multiplied by its weight and it is filled in. We have:

```
  8   3   18     0     0   0   ←→ C
 16   3  300  2000     0   0   ←→ B0
 21   0  100    20   100  20   ←→ B1
 21   0   20     1    20   1   ←→ B2
100  20                       ←→ S
  2                           ←→ T
```

Next, the scalar values are found and moved out of the loop by adding them together the scalar values are multiplied by the weights (i.e., +/×/*CB*[*I*;1 3] for *I* denoting rows which contain scalar values. Finally, we discover adjacent AP vectors which are of the form:

CB[*I*-1;5]←→ ×/*CB*[*I*;4 5]

This condition occurs in the rightmost two columns of *B*1 and *B*2 which are of the form:

```
. . . 100  20
. . .  20   1
```

This means that the least significant subscript moves in steps of 1 to a count of 20 then that counter is reset while the next counter moves in steps of 20 to a count of 100, and the *B*2 counter steps through 20 increments between each change of the *B*1 counter. Thus, a single counter using steps of 1 up to 2000 can be substituted. After these two steps are accomplished there will be some unused blocks (rows of *CB*). Active blocks are moved to the head of the list.

```
      ∇INDB[□]∇
    ∇ INDB;D;W;S;K;F;Q
[1]   ⍝ NOTE THAT 0 ORIGIN IS USED
[2]   'RANK' ERRORIF L[1]≠⍴⍴A
[3]   D←(⍴A)[Q←¯1+⍴⍴A]
[4]   S←W←1+F←0
[5]   L[K]←L[K←L[2]]+16
[6]   ⍝ START AT INNER BLOCK AND PROCESS
[7]   ⍝ ONE BLOCK AT A TIME
[8]  LOOP:L[K+ 2 3]←D,W
[9]   →(T0,T1,T2, 0 0 ,T5)[8|L[K]]
[10] T0:F←F+W×CHECK L[K+1]-IORG
[11] MOR:S←S×L[K+4]
[12] MOR2:→ENDLOOP IF L[K←K-6]>7
[13]  W←W×D
[14]  →LOOP,D←(⍴A)[Q←Q-1]
[15] T5:→L1,L[K+4]←L[L[K+4]]←D
[16] T1:F←F+W×CHECK L[K+1]
[17]  →(0×CHECK L[K+1]+L[K+5]×L[K+4]-1)/L1
[18] L1:L[K+5]←W×L[K+5]
[19]  →MOR IF(L[K]>16)∨1≠2|L[K+6]
[20]  ⍝ SUCCESIVE AP VECTORS,REDUCE IF POSSIBLE
[21]  →MOR IF L[K+5]≠L[K+10]×L[K+11]
[22]  L[K+6]←L[K+6]-4|L[K+6]
[23]  S←S×L[K+4]
[24]  →MOR2,L[K+ 5 4]←L[K+11],L[K+4]×L[K+10]
[25] T2:L[P]←W×CHECK L[P←L[K+5]+⍳L[K+4]]-IORG
[26]  →MOR
[27]  ⍝ NOW MOVE ACTIVE BLOCKS TO HEAD OF LIST
[28] ENDLOOP:P←6+K←0
[29] L4:→L2 IF 0=4|L[K←K+6]
[30]  →L3 IF K=P
[31]  L[P+⍳6]←L[K+⍳6]
[32] L3:P←P+6
[33] L2:→L4 IF~L[K]>15
[34]  L[6]←8+8|L[6]
[35]  L[P-6]←16+16|L[P-6]
[36]  L[0]←F
    ∇

      ∇EVAL[□]∇
    ∇ Z←EVAL;I;J;N;T
[1]   ⍝ NOTE THAT 0 ORIGIN IS USED
[2]   →NOTNULL IF 0≠J←×/N←¯1↓(-1+¯1↑L)↑L
[3]   Z←N⍴1↑,A
[4]   →0
[5]   ⍝THIS SETS I=ZERO OR BLANK
[6]  NOTNULL:I←1↑(T←0)⍴A
[7]   ⍝ SO NOW WE CAN INITIALIZE Z
[8]   Z←J⍴I
[9]  LOOP:→SKIP IF 0>I←INDC T>0
[10]  Z[T]←(,A)[I]
[11] SKIP:→LOOP IF J>T←T+1
[12]  Z←N⍴Z
    ∇
```

Figure 3 INDB and EVAL

```
      ∇INDC[□]∇
    ∇ Z←INDC T;U;S;COUNT;STEP;W
[1]   ⍝ NOTE THAT 0 ORIGIN IS USED
[2]   →FETCH IF T
[3]   ⍝INITIALIZE LOOPS AND GET FIRST VALUE
[4]   Z←L[0]
[5]   L[1]←U←0
[6]  LOOP:S←256|L[U←U+6]
[7]  LOOP1:STEP←L[U+5]
[8]   COUNT←L[U+4]-1
[9]   →(DOWN00,DOWN01,DOWN10)[2⊥BIT 6 7]
[10]  ⍝ CAN ONLY OCCUR IF 0=⍴⍴A
[11] DOWN00:→0
[12]  ⍝ AP CASE
[13] DOWN01:→(0≤L[U+3]←Z)/POS
[14]  STEP←0
[15] POS:→(BIT 3)/INNER
[16] NOTIN:→LOOP,L[U+ 1 2]←COUNT,STEP
[17] INNER:U←1+4×U
[18]  L[4]←Z
[19] INNER1:→0,L[1 2 3]←U,COUNT,STEP
[20]  ⍝ GENERAL VECTOR OR ARRAY
[21]  ⍝ 'STEP' ACTUALLY CONTAINS OFFSET
[22] DOWN10:W←Z
[23]  Z←(¯1,Z+L[STEP])[L[STEP]≥0]
[24]  →INNER2 IF BIT 3
[25]  →NOTIN,L[U+3]←W
[26] INNER2:U←3+4×U
[27]  →INNER1,L[4]←W
[28]  ⍝
[29]  ⍝ GET NEXT VALUE,L[1 2 3] CONTAINS
[30]  ⍝ COUNT, STEP OR OFFSET,OLD VALUE
[31] FETCH:→NONSC IF 2|S←256|L[1]
[32] FINAL:'NO MORE ELEMENTS IN LIST'
[33]  →Z←0
[34] NONSC:→ENDS IF 0>L[2]←L[2]-1
[35]  →VEC IF BIT 6
[36]  →0,Z←L[4]←L[3]+L[4]
[37] VEC:→0 IF 0>Z←L[3]
[38]  →0,Z←L[4]+L[L[3]←L[3]+1]
[39]  ⍝ INNER LOOP IS FINISHED,TRY NEXT OUTER
[40] ENDS:S←256|L[U←⌊L[1]÷4]
[41] UP:→FINAL IF BIT 4
[42]  S←L[U←U-6]
[43]  Z←L[U+3]
[44]  STEP←L[U+2]
[45]  →UP IF 0>L[U+1]←COUNT←L[U+1]-1
[46]  ⍝ OUTER IS OK,SO RE- CYCLE INNER
[47]  →DOWN1 IF BIT 6
[48]  →LOOP,L[U+3]←Z←Z+STEP
[49] DOWN1:STEP←L[U+2]←STEP+1
[50]  →LOOP,Z←(¯1,Z+L[STEP])[L[STEP]>0]
    ∇
```

Figure 4 INDC

When we finish we have L of the form:

```
6000   g  g  g     g  g
  29   g  g  g  2000  g
   g   g  g  g     g  g
   g   g  g  g     g  g
 100  20
   2
```

The g's denote garbage or unused entries. At the end of the optimization L has the form C, $B0$, $B1$, ..., BK, V, S, T where

$C \leftarrow F$ g g g g g

$BI \leftarrow X$ g g g N S

$V \leftarrow$ (old values - $IORG$) × weight factor

$S \leftarrow$ shape of result

$T \leftarrow$ rank of result

and g still denotes garbage entries.

If $4|X \leftarrow \rightarrow 0$ then block no longer used

1 then block generates $S \times \iota N$ (i.e., AP or missing subscript)

2 then block generates $L[S + \iota N]$

F is the fixed part of the addressing from scalars or the offset $START$ in $START + STEP \times \iota NUMBER$

The routines $EVAL$ and $INDC$ are used to pick up the values. The structure I containing the active BI uses, $BI[1]$ as a counter to step through ιN;$BI[2]$ is used as an offset or step; and $BI[3]$ holds the currect value.

3.2.2 Discussion

Several comments about Hassitt and Lyon's indexing technique [11] which we have sketched above are appropriate at this point.

First, the number of multiplications encountered in the translations of the index by the polynomial is greatly reduced. In $A[S1;S2;S3]$ we expect $(^{-}1 + \rho\rho A) \times (\rho,S1) \times (\rho,S2) \times (\rho,S3)$ multiplication for example for $\rho A \leftarrow \rightarrow 300\ \ 100\ \ 20$ and $A[3\ ;\ ;]$ this would be $2 \times 1 \times 100 \times 20$ or 4000. The method described above turns out to use two for each AP vector and one for each scalar and so for the example given there are 5 multiplication. In general at most only $(\rho,S1) + (\rho,S2) + (\rho,S3)$ multiply operations are needed. AP vectors (and therefore missing subscripts) are effort saving in terms of multiply operations.

Next, the routines are straightforward and may be written using mostly LOAD, STORE, and BRANCH operations. Even with complex functions such as RESIDUE the routines were designed so that in $A|X,A$ is a power of two. The function can be implemented by MASKING to pick up the $2 \circledast A$ low order bits of X. Thus, the routines are extremely amenable to be implemented in a microprogram form.

Finally, the block L describing the indexing is separated from the values and since the calculation take place in L only enough additional space to store the local variables of the routines will be required to make re-entrant coding of this approach directly available. In terms of storage for the indexing function if all index expressions are scalar, or AP vectors (which include missing subscripts), then $(6\times(1\times\rho\rho ARRAY)) + 1+\rho\rho RESULT$ or less than $7\times1+\rho\rho ARRAY$ cells are required,

and in general the space for V must be added. In arrays of large rank there may well be adjacent subscripts in which there are missing expressions. These may be squeezed together to reduce the number of loops and hence the overhead of loop initialization, incrementation and testing for the end condition

3.3 Other Storage Mappings

In the preceding section polynomial indexing was used to map and n-dimensional rectangular array onto linear storage addresses. Some time ago (1962), S. A. Hoffman [12] discussed ways of defining, allocating, and referencing data structures which generalize rectangular arrays. Subsequent to Hoffmans work, P. Deuel [13] published a more efficient algorithm for the storage mapping function of Hoffman.

Before examining an APL version of Deuel's algorithm, we exhibit an example of a structure to which such a mapping function may be applied. Suppose we have a personnel file having three levels. The number in parentheses denote the number of instances of that item. In brackets we give the name of the item.

Personal Record (40) [P]

- Name (20) [U]
- Salary History (10) [S]
  - Date (6) [D]
  - New Salary (5) [T]
- Evaluation List (10) [E]
  - Date (6) [D]
  - Rating (2) [R]

Every structure is either an array of (unstructured) particles or a (sub)structure of identical instances. It is assumed that all particles are alloted che same number of storage units whether the units be bits, bytes, half words, words, or some larger unit of storage.

In this example there are 40 instances of Personel Record; each instance is a sequence of structures:
< Name, Salary History, Evaluation List >. Name consists of 20 particles and the structure Salary History has 10 instances each instance consists of the Data (which has 6 particles) and the New Salary (having 5 particles).

Table 2 gives a tabular formulation of the same structure.

Name	N	P	U	S	D	T	E	D	R
Level	L	1	2	2	3	3	2	3	3
Court	C	40	20	10	6	5	10	6	2

Table 2. A Tabular description of a structure

A reference expression is a sequence of a subscripted names in the form of either

$$A_1(x_1)A_2(x_2) \ \ldots \ A_k(x_k)$$

or

$$A_1(x_1)A_2(x_2) \ \ldots \ A_k$$

with $\text{level}(A_1) < \text{level}(A_{i+1})$ and with $\text{level}(A_1) = 1$ and with each $0 < x_i$ and $x_i \leq \text{count}(A_i)$. The first form designates a specific instance or particle and the second references a specific (sub)structure. Thus, P(5) denotes the 5th instance of P and P(5)U denotes the name of P(5).

P(5)S(2)D(3) gives the 3rd instance of S within the 5th instance of P. Note that there is no confusion as to which D is referenced because the sequence of names is unique.

Deuel's algorithm is given in Figure 5 as an APL function. The input is a matrix of two rows and as many columns as there are columns in the tabular form of the structure. In fact the input is the tabular structure excluding the name row. Table 3 shows the output from the Storage Mapping Function, *SMF*, rejoined with the names. Table 3 is thus Table 2 with rows Q and M appended.

NAME	P	U	S	D	T	E	D	R
LEVEL	1	2	2	3	3	2	3	3
COUNT	40	20	10	6	5	10	6	2
Q	0	0	20	0	6	130	0	6
M	210	1	11	1	1	8	1	1

Table 3. Q and M appended to the tabular form of a structure.

The values for Q and M may be used to calculate the storage location is $A_1(x_1)A_2(x_2) \ldots A_k(x_k)$ relative to the base address for A_1 by

$$\sum_{i=1}^{k} [Q(A_i) + M(A_i)(x_i - 1)]$$

Relative to the example structure introduced above the particle P(5)S(2)D(3) address, in terms of an offset from the start of storage, is calculated as:

```
      ∇SMF[□]∇
    ∇ Z←SMF LC;Q;M;W;I;LIMIT;N;J;T
[1]   N←¯1↑⍴LC
[2]   Q←N⍴0
[3]   M←N⍴1
[4]   →NO∆END IF N≠1
[5]   →EXIT AFTER M[1]←1
[6] NO∆END:W←0
[7]   I←1
[8] SCAN:I←I+1
[9]   →BRANCH IF I≤N
[10]  →FIRSTM AFTER LIMIT←LC[1;1]
[11] BRANCH:→(LINK,DEFINE,UNLINK)[2+×-/LC[1;I- 1 0]]
[12] LINK:M[I-1]←W
[13]  W←I-1
[14]  →SCAN AFTER Q[I]←0
[15] DEFINE:M[I-1]←1
[16]  →SCAN AFTER Q[I]←Q[I-1]+LC[2;I-1]
[17] UNLINK:LIMIT←LC[1;I]
[18] FIRSTM:M[I-1]←1
[19]  J←I-1
[20] MORE:T←M[W]
[21]  M[W]←Q[J]+LC[2;J]×M[J]
[22]  J←W
[23]  W←T
[24]  →MORE IF LC[1;J]>LIMIT
[25]  →EXIT IF I>N
[26]  →SCAN AFTER Q[I]←Q[J]+LC[2;J]×M[J]
[27] EXIT:Z←LC,[1] Q,[0.5] M
    ∇
```

```
      ∇IF[□]∇
    ∇ Z←L IF R
[1]   Z←R/L
    ∇
```

```
      ∇AFTER[□]∇
    ∇ Z←L AFTER R
[1]   Z←L
    ∇
```

Figure 5 The Structure Mapping Function, SMF

$$[0 + 210.4] + [20 + 11 \cdot 1] + [0 + 1 \cdot 2]$$

$$= 840 + 31 + 2 = 873$$

the case of

$A_1(x_1)A_2(x_2) \ldots A_k$ implies the sequence of $A_1(x_1)A_2(x_2) \ldots A_k(x_{k_i})$ for all x_{k_i} ranging in sequence from 1 to count(A_k). Thus $A_1(x_1)A_2(x_2) \ldots A_k$ and $A_1(x_1)A_2(x_2) \ldots A_k(1)$ point to the same element and since particles in a substructure are stored in consecutive elements, we may address the first element and then pick up the correct number of elements and that number is given by $M(A_k) \cdot C(A_k)$.

An examination of the access function parameters produced by *SMF* indicates that the structure is traversed left to right listing each of the particles in lexigraphic order of counting in the names.

While the above looks something like an indexing operation in Origin 1 for arrays considered earlier, it is perhaps not entirely clear how *SMF* will deal with an array having the regularity found earlier.

3.4 Arrays of Structures

If we consider a three dimensional array, *A*, of say 5 by 4 by 3, then we may take the array to be a structure which consists of 5 planes each of which has 4 rows, each of which has 3 elements. In tabular form this gives:

NAME		Planes	Rows	Columns
Level	⟷	1	2	3
Count	⟷	5	4	3

If we use this as data for *SMF* we obtain:

```
Q ←→  0    0    0
M ←→ 12    3    1
```

Clearly since every name must be present if we index in all coordinate positions (with scalars in an APL sense), the form

$$\sum_{i=1}^{k} [Q(A_i) + M(A_i)(x_i - 1)]$$

reduces to *+/Q+M×X-IORG* and since *Q* ← → 3ρ0 and *M* is identical to the *W* encountered previously, the polynomial indexing of regular arrays is a special case of the structures mapped by *SMF* as shown in Figure 5.

The equivalence *A*[*I*;*J*;*K*] ← → Planes(I)Rows(J)Columns(K) thus holds, and further the equivalent descriptions given below also follow

A ← →	Planes
A[*I*;;] ← →	Planes(I)Rows
A[*I*;*J*;] ← →	Planes(I)Rows(J)Columns

However, there are no equivalents to the forms: *A*[;*I*;], *A*[;;*I*], *A*[*I*;;*J*] or *A*[;*I*;*J*].

Of course we can always simulate these missing forms by scalar indexing, stepping through the appropriate set of subscripts in the spirit of the previous section, and reshaping the result. It should be noted that the last dimension of the array occurs as a particle array whereas the other dimensions of the array refer to structures. If we use the tabular form,

```
L ← → 1    2    3    3    3
C ← → 5    4    1    1    1
```

we see that the last dimension is now described as a structure, each having a single particle. The names and the result of *SMF* are

NAMES	Planes	Rows	Col1	Col2	Col3
Level ← →	1	2	3	3	3
Count ← →	5	4	1	1	1
Q ← →	0	0	0	1	2
M ← →	12	3	1	1	1

Two things are to be noted: (1) The weights in *M* in the first two coordinates positions are not disturbed; and (2) To index in the third dimension requires the selection of the proper name in the third level of the structure. Thus, *A*[3;2;2] is equivalent to Planes(3)Rows(2)Column2.

If we consider using this same artifice on the second dimension, we get a structure which looks like:

L ← →	1	2	3	2	3	2	3	2	3
C ← →	5	1	3	1	3	1	3	1	3
Q ← →	0	0	0	3	0	6	0	9	0
M ← →	12	3	1	3	1	3	1	3	1

The cost, in terms of the size of the descriptors *Q* and *M*, has increased, and in the above we see that the last dimension is still encoded as a particle array in the structure. To alleviate this, as in the immediately preceding, would require an additional $8 \leftarrow \rightarrow (3 \times 4) - 4$ elements in *Q* and *M*. As other than the last dimension is altered in the fashion suggested above, the size of the descriptor matrix, *QM*, grows prohibitively large. If we alter the structure in the last dimension only, then the *QM* descriptor of the structure for *A* is 2 by

$(\rho A)[(\rho\rho A+IORG-1] + {}^{-}1+\rho\rho A$ matrix.

Actually since the first ${}^{-}1+\rho\rho A$ elements of Q are 0's and the last $(\rho A)[(\rho\rho A)+IORG-1]$ elements of M are 1's, the space requirements may be cut in half if we know the rank of A. Thus Q and M can be a vector, *VQM*, with *VQM* given by: $VQM \leftarrow ((^{-}1 + \rho\rho A) \uparrow M),(^{-}1+\rho\rho A)\downarrow Q$ and for the example shown on the top of page 37 we have $VQM \leftarrow \rightarrow$ 12 3 0 1 2. It is straightforward to return to array indexing when we consider the structure formulation. Consider, *AR*, indexed by a scalar in each co-ordinate position, is the meta-notation of Abrams [10].

AR[;/*INDEX*] where *INDEX* is the catenation of the $\rho\rho AR$ scalar values. Then if we construct L and C for *SMF* as

$L \leftarrow (\iota^{-}1 + \rho\rho AR), (\rho AR)[\rho\rho AR]\rho\rho\rho AR$

$C \leftarrow (^{-}1 \downarrow \rho AR), (\rho AR)\ [\rho\rho AR]\rho 1$

in Origin 1 (since that is required in the next step, where SMF is called).

Next construct *VQM* from Q and M, the output of *SMF*. The index is then given by

$(,AR)[IORG+((^{-}1+\rho\rho AR)\downarrow VQM[^{-}1\uparrow INDEX]+(^{-}1+\rho\rho AR)\uparrow VQN)+.\times(^{-}1\downarrow INDEX)-IORG]$

3.5 Application of Structure Mappings to the Array Model

Up to this point our examination of structures as an alternate mapping scheme has not seemed to be directly applicable. However if we return to the 5 by 4 by 3 array introduced earlier, and consider the case where in the third dimension the elements do not take up the same amount of space, but rather let count at the lowest level denote the

amount of space required by each particle. Suppose the space required for the three elements is 2, 1, and 2 units respectively. The structure becomes:

```
L ← →  1  2  3  3  3
C ← →  5  4  2  1  2
Q ← →  0  0  0  2  3
M ← → 20  5  1  1  1
```

and

```
VQM ← → 20  5  0  2  3.
```

This will allow us to calculate the starting point in the raveled array of any item. A little thought will show that the lengths (in addressing units) are also needed and this can either mean that the information in Count is retained or an additional element can be appended to *VQM* so that the length (in addressing units) can be obtained by the difference of two adjacent elements of *VQM*. (After we have dropped the first ¯1+⍴⍴4 elements.)

Thus, the structure model provides a means of handling the varying lengths of data encoded in a frame of telemetry data. The non uniformity of data is solved providing: (1) the addressing units are small enough to prevent excessive wastage; (2) the experimental data is unpacked to the extent that it is on addressing boundaries; and (3) the data layout is such that we may place these varying lengths in the last dimension. The validity of the first assumption depends on

(1) The addressing structure of the computer. If the computer can address to the bit, little overhead in space will be

encountered. If only full words can be addressed, then a great amount of waste can occur. Assume that the word length is 32 bits with byte (8 bit) and half-word addressing available.

(2) The statistical distribution of the lengths, in bits, of the experiments on board a particular spacecraft. Assume that the range is from 6 to 32 bits. (Reference [9] p. 5). We take the frame size to be in the order of 1200 bits [6,7] and that on the average a data point is 10 bits [6]. This amounts to approximately 120 experiments, each of which requires 2 bytes when aligned on byte boundaries: $2 \leftarrow \rightarrow \lceil 10 \div 8$. Thus a frame takes up 240 bytes rather than 150, for a 60% increase. This neglects an intermediate approach of arranging only part of the data on the boundary and doing so with experiments which are judged as being more likely as receiving significant activity. The remaining data is left packed, to be unpacked in demand. Because of the number of bits usually used in A-to-D conversion, we take the large bit lengths to be unlikely.

We also assume that positional and spacecraft attitude data requires four parameters (including time) for each of the two calculations. Four bytes for each parameter are assumed [6], for a total of $4 \times 2 \times 4 = 32$ bytes. This data is generally stored separately and it appears by examination that it is not unusual for such calculations to be made with a frequency of approximately one per minute during the life of the spacecraft. We assume that there are interpolations applied to the time to obtain a corrected position from the spacecraft position data. To

add the data for a frame, assuming the bit lengths of the experiments are sufficient to make the time, and hence the positional, corrections, requires 32 bytes (13.3% of the frame) for the positional data.

At the same time perhaps only 1% or less of the data is expanded in the sense of calculating the position and/or attitude data. On the one hand having the positional data in line makes it possible to include its retrieval in the same data movement as that required for the rest of the data. The small amount of data using orbital or attitude data relative to the total volume suggests that such an approach would not be taken. One alternative would be to have several frames blocked together and attach positional and orbital data to that ensemble in such a fashion that interpolations can be made. A more usual approach would to provide a link which will mark whether the orbital/attitude data has been computed for those frames and if so, provide its location in the address space.

Thus, the array might be viewed as organizing the data in the following way:

DATA[*STATION*;*EVENT*Δ*SOURCE*;*FRAME*Δ*NOS*;*FRAME*Δ*EVENTS*]

This places all of the data of a single frame in the last dimension so that the encoding scheme just outlined can be applied. The next to last coordinate is labeled *FRAME*Δ*NOS*, and we note that while each frame has an index number denoting the sequence in which the data was generated, the use of that number (or some sequence of them) is inconvenient to the user. Data associated with a particular longitude and latitude or with some orbital event such as the perigee is more likely to be the way in which a user thinks of defining his data search. Values for such a

search are not directly associated with the frame numbers but rather with data itself which is logically a part of the frame. The fact that this data is actually stored elsewhere, such as in the orbital/attitude data tables, is a system convenience for storage efficiency.

The array model need not be made to suffer however. The indices can be arranged to suit the needs of the user community (with perhaps a loss of retrieval efficiency) by making the index expressions functional so that they generate the indices indirectly.

The fact still remains that the way archival storage has been handled in the past, and to a large extent continued in the modifications suggested by this proposal, makes concessions to the data volume and not to the direct needs of the users. Most of the actual use of the data has been up to the individual experimenter.

4.0 THE USER-DATA BASE INTERFACE

The previous sections have examined the use of the array as a tool to interface the scientific user to a large data base. The particular APL-like flavor of the array need not be retained however.

The usual structure used to contain the data is often one in which the needs of the end user are secondary to the requirements of collecting and storing the information. Practice has been one of viewing the storage as a repository of collected data rather than the user view of data which is: "The information is useful if it is mine or if it relates to my problem and it is taking up resources otherwise". In this second viewpoint the data base becomes more inverted and in some respects similar to a management information system.

A typical management information system requires that the user be able to insert, move, and retrieve information. Hence any data manipulation algorithm must have ways of finding inverses and must necessarily be cumbersome. The usual approach is to use some variation of tree and pointer algorithms, sometimes in conjuction with an indexed or table lookup procedure where a new table is written for inverse operations. These are space and time consuming operations. See Reference [14] for a complete discussion of such strategies. The problem is complicated by the fact that many of NASA's requirements are unique.

If it is the goal that the user can browse and retrieve information from large data bases on-line, then we note that the experimenter never inserts new information into the original file. Similarly, the data base has the unique property that new information is only appended to it but never inserted. In a sense this is not quite true. For example orbital/attitude data may be inserted later because it is not available at the time of writing the (on-line) data base. However, the volume is usually small compared to the rest of the data. In such a case the space for the pointer chains can be provided for most of the stored information then the space can be reserved at the initial storage time and the data base then behaves as though no insertions are made. Attempted retrieval would only indicate that the information is unavailable. This property allows for very powerful retrieval algorithms which will not be unnecessarily complex. Because an inverse operation is never necessary, the information in the data base can be compressed, enhanced, or only certain features extracted without any loss in generality but with the

corresponding advantage of having to maintain a smaller data-base then the case where a more general data structure is required.

Certain problems must always be considered in designing a data base system. We will attempt to touch on most of these problems in the following disucssion:

1) How should the data be stored? Should the data be sequential, linked, or indexed or in what combination? Are there better ways of storing data? Do we assume that data will always be physically stored as a string? Are inverses necessary? Is decoding and its inverse encoding necessary? Must we worry about inverse operations for feature detection, compression, and enhancement?

2) How should the system look to the user? What primitives must he have available to him? What type of language should the user have to learn? Is the language easy to learn and yet is it powerful? Does it have or can the user easily create all the features that he would need?

3) How should the data be manipulated? What algorithms must be implemented into hardware? Software? Are these algorithms independent of how the data is stored?

Questions under 1 are answered in part by the use of the array as a means of adapting the data base to the user without expending a great deal of effort in inverses. For example in an ERTS data base situation a user might be interested in:

MOST RECENT PICTURE AT LONGITUDE X AND LATITUDE Y WITH LESS THAT 1% CLOUD COVER

One can imagine that if the array model has dimensions for longitude,

latitude and time, the need for an associative search is less. However if a request is for

ALL FRAMES SHOWING SHORE LINES OF LAKES LARGER THAN 20 SQUARE MILES,

then the search is bound to be difficult because the data sought is associated with some search criteria applied to the data values. However if the search can be bounded (possibly by inference) the task is reduced.

Questions 2 and 3 relate to requests such as the above. Further, languages such as GOTRAN [8] or the processing language suggested by Broglio [9] are useful but are at the other end of the problem.

While this effort is too restricted to presume to design a user query language, the following philosophy should prevail. The system must be oriented to the human user who should be able to examine the contents of the file without having the need to know how the machine manipulates or stores the data. This comment is expanded as follows:

1) A user should be able to examine a file without knowing its structure. The user should never have to worry about differentiating between a scalar, vector, or array; he also should never have to worry about how the data is physically stored.

2) A user should be able to examine a file without knowing its representation. A user should never have to worry about things like: Does an element represent character? Numeric (Boolean, integer, real)?

3) The file should be presented to the user in a form he normally expects to see it. That is if a particular file is usually thought of as being an array, it should be presented to the user as that array

rather than as its ravel.

4) Even though it is not achievable, the system should be designed to try to be all things to all users. Therefore the system should have built in defaults which can be overridden by different users with different experience and different authorization. For example an inexperienced user would like the convenience of working in a question-answer mode; and experience user would find such a system too much of a nuisance and would rather type his instructions directly.

5) The language should have a facility for report generation, representing and restructuring the file in any desired form for either visual representation or for storage in another medium.

The following characteristics are probably desirable:

1) Each command sequence shall normally be parsed, incrementally compiled, and executed after each carriage-return. Sequence of commands should be allowed to be collected for execution.

2) No user defined iterations or recurrsions should be allowed, but enough power should be included that this is not restrictive.

The language must be mathematically sound; the language should essentailly be a set-relational language that will manipulate actual data only at the very end to eliminate excess number crunching. The language must have the following features and primitives:

1) Set Definer: The user should be able to define symbols.

SetA ⊂ Jim, Bob, Mary

SetB ⊂ "All mountains over 3 miles"

SetC ⊂ SetA, Jane

2) Set Compressor: Sets should automatically be compressed and stored to save storage and time.

(A B 5 B 5 5)← →(A B 5)

3) Symbol/Boolean/Relational/Functional/Group/Logical Manipulators: This should be automatic

~(~A ∪ ~B) ← → A ∩ B

4) Precedence of Operations: One is tempted to require that any expression must be automatically be interpretted to have natural precedence.

× before +

~ applied only to symbol next to it unless overridden by use of parentheses

The difficulty is that for many proposed functions precedence is arbitrary. Hence we suggest a scheme similar to that of APL.

5) Symbol Evaluator: Expressions should be automatically evaluated to its simplest form.

A ∪ B ← → Jim, Bob, Mary, Jane

A ⊂ C ← → TRUE

6) Primitive Operations: The language must have at least the following primitive operations:

Logical: All, ∩, ∪, ~, ⊂, −, <, >, =

Arithmetic: +, −, ×, ÷

7) New Operation-Assigner: The user should be able to define new operations.

A □ B <-- All − A ∩ B

8) Number-Assigner: The user should be able to assign dimensioned numbers.

Bob ← 3.63 inches

9) Locator-(Feature-Detector): The user should be able to describe his data.

All lakes over 2 miles wide

10) Number/Arithmetic Manipulator: The machine should attempt to optimize by simplification at the source.

A + B - B ← → A

Note the usefulness of this if B ← ι10*E*6

11) Conversion Declaration: The user should be able to define his own conversions that the system would automatically use.

12 in. ← □ → 1 ft.

12) Number Converter-Calculator: Automatic conversion of dimensioned numbers. Whenever conversion has been defined.

1.5 ft. × 2 in. ← → in feet

36 sq. in. ← →

0.25 sq. ft.

13) Report-Generator: Printing should automatically be done in a way that the user would normally expect to see it. Control is placed in the hands of the user allowing a variety of formatting.

5.0 IMPLEMENTATION CONSIDERATIONS

We have assumed that input and storage processing are done as current plans and practice require. The data is assumed to be placed in the array model format with whatever modifications required in the last (frame stream) dimension to accommodate packing of the different sized formats that may exist for encoding the experimental data. A further requirement is that the data is aligned on machine addressing boundaries.

Finally, enough low order coordinate dimensions are introduced to strike a balance between making the user requests easier to fit into the array indexing scheme and expanding the storage requirements due to holes in the array.

To illustrate this last trade-off consider the data collected by ERTS. Since the data relates to positions on the surface of the earth, it is reasonable to introduce the coordinate indices of longitude and latitude as two dimensions. However, since the data collected over any given intersecting band of longitude and latitude is small relative to the entire volume of data, the array model will have holes in those dimensions. Moreover, the areas are not of equal size (even in approximation); there is surely more data to be collected over North America than over the middle of the ocean.

Still, the use of the array model suggests a number of considerations in implementations; and while the extent of this study is not sufficient to allow a full investigation, a number of points should be noted.

The alignment on addressing boundaries is a requirement which

expands the storage space needed in order to make demand decommutation more tactable. A computer capable of addressing to the bit level will remove the need for excess backing store overhead. Also there is likely to be some degree of variability if the bit lengths of the experimental data over a collection of satellites. Thus a computer which is bit addressable and deals with bit vector lengths over a reasonable range would be advantageous. A structure such as the Burroughs B1700 supports requirements such as these.

The indexing routines as explained previously are easily implemented in microcode. The same is true for the array modifications introduced to handle the different number of bits needed to encode the various experiments data. A great deal of the overhead which would be associated with an essentially interpretive command structure on an interactive system may be absorbed in microcoded routines.

The requirements indicate that a (collection of) mini computers can be configured in a (multi-) processing system for this problem. The bottleneck is then the data base channel.

Virtual memory using a hardware supported paging mechanism is not necessarily mandated by the considerations undertaken here. When indexing in the array model takes place, several pieces of information are known. The size of the result is known before the values are fetched. The locations of the values can be determined from the indices during the analysis phase of indexing and the sequence for fetching the data from backing store is known and can be optimized. As each segment of backing store data is accessed to obtain the required data

the space can be returned to a buffer area. Thus, if the address space is large enough to hold the data base, the indexed array model permits a variety of mapping mechanisms.

Finally, it should be pointed out that in data volumes of the sort considered in this report there is a high channel bandwidth required if the following all hold: (a) output decommutation is on a demand bases (b) data specified is scattered through significant fraction of the data base, and (c) only a small portion of each segment (such as a frame) is required. Under such conditions even if the data can be located easily within the segment, the volume still causes problems. An appreciable number of these units must be brought from storage and one segment read request rapidly follows another since there is little work to be done on each.

Such activity on demand precludes increasing the processing efficiency by dealing with all requests relating to each segment. Vector machines such as Control Data's STAR or Texas Instruments ASC directly confront such large data movement problems. In fact the kernels encountered in the array indexing model introduced here, where the structure is mapped onto linear address space and within such a vector appropriate pieces of data occur at regular intervals, are well suited to vector oriented pipelining. While a great deal of data is moved, the system knows the amounts and logical locations of the pages and hardware mechanisms can continue to fetch pages until main storage is filled. Since there is usually an assumption that a small portion of a page is extracted, page activity soon exhausts a page making it a candidate for

release by task A since it will not be needed again by A (although task) B may require the same data, causing problems). Even under such circumstances it is sometimes more advantageous to pass over the data twice, once testing and marking the data and the second time actually selecting it by compression. Some of the effects of using vector oriented machines are discussed by J. L. Owens [15]. While the details of his examinations can not be related precisely to the spacecraft data problem, the mapping of large sets of data onto linear representations and treating the results as vectors, does apply.

Thus, it seems to be quite likely that not only does the data array appear to be a useful one for spacecraft data but also a vector pipelined machine architecture for such problems may be a natural choice.

References

1. *APL*\360 Users Manual, A. D. Falkoff and K. E. Iverson, IBM Corp., GH20-0683-1, 2nd Edition, March, 1970.

2. *APL*\360 Reference Manual, S. Pakin Science Research Associates, 2nd Edition, 1972.

3. *APL*\360 An Interactive Approach, L. I. Gilman and A. J. Rose, J. Wiley and Sons, 1970.

4. *APL* A Short Course, S. Pakin, Prentice-Hall, 1973.

5. ADP Application Study (Feasibility Study)for A Central Data Handling Facility and Associated Remote Terminals for Atmospheric Explorer, C, D, and E, Goddard Space Flight Center, Greenbelt, MD, September, 1971.

6. Telemetry On-Line Processing System (TELOPS). ADP Feasibility Study, September, 1971, Informatics, Inc., TR-71-1500-05 for GSFC under NASA Contract NAS5-20240.

7. Telemetry On-Line Processing System (TELOPS) System Specifications, November, 1971, Informatics, Inc., TR-71-1500-07 for GSFC under NASA Contract NAS5-20240.

8. The Data Reduction Laboratory Reference Manual, B. A. Walton, J. J. Quann, F. A. Keipert, January, 1969, Goddard Space Flight Center, Greenbelt, MD, Document X-565-69-56

9. A New Approach to Telemetry Data Processing, Carlo J. Broglio, May, 1973, Goddard Space Flight Center, Greenbelt, MD. Document X-522-73-135 (Ph.D. Dissertation, University of Maryland).

10. An APL Machine, Philip S. Abrams, Ph.D. Dissertation, Stanford University, 1970 (SLAC Report 114, February, 1970, Stanford Linear Accelerator Center).

11. "Efficient Evaluation of Array Subscripts of Arrays", A. Hassitt and L. E. Lyon, IBM Journal of Research and Development, 16, 1 (January, 1972) pp. 45-57.

12. "Data Structures that Generalize Rectangular Arrays", Samuel A. Hoffman, AFIPS Joint Computer Conference, Spring 1962, pp. 325-333.

13. "On a Storage Mapping Function for Data Structures", Phillip Deuel, Communications of the Association for Computing Machinery, 9, 5 (May, 1966) pp. 344-347.

14. File Structures for On-Line Systems, David Lefkovitz, Spartan Press, 1969.

15. "The Influence of Machine Organization on Algorithms", J. L. Owens, in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, Ed., Academic Press, 1973.