Emulation of an APL machine on a System/370 is exemplified by the APL Assist, a microprogram that enables APL expressions and defined functions to be executed at the hardware level. This paper discusses what the APL Assist does, how it works, and the way it interacts with System/370 software. Execution times for APL programs with and without the Assist are compared.

An APL emulator on System/370

by A. Hassitt and L. E. Lyon

There are several methods of executing programs written in a high-level language. The most widely used is to compile the programs into machine language. Another is to translate the programs into some intermediate form and then execute that form interpretively. A third method is to build a machine to directly execute either the high-level language or the intermediate form.

APL programs are executed by interpreters, as it is difficult and perhaps impossible to compile APL.[1] The interpreters are successful, but there are several reasons for wanting to build a machine that would execute APL directly. Such a machine could offer the possibility of a radically new architecture, it could affect the way software is written and developed, and it could speed APL execution.[2]

An APL machine would differ from a conventional machine, like a System/370, in that a conventional machine executes instructions. For example, the instruction A 2,B causes the word at location B to be added to the word in register 2. The machine knows nothing of the properties of the words; it simply performs integer addition because the instruction specifies it. APL programs, however, do not contain instructions; they contain expressions, statements, and functions. The expression $B+C$, for example, is merely a request for the addition of B and C; the machine would have to determine the following:

- whether the addition is allowed (for example, the result would be a domain error if either B or C were an array of characters);
- whether integer or floating point addition should be used;
- whether one or both operands should be converted to floating point form;
- whether the operands are conformable (for example, if B is a 10-by-20 matrix, then C must also be a 10-by-20 matrix, or else it must contain only one element);
- how many additions are required (for example, if B is a 10-by-20 matrix and C is conformable, then 200 additions are required).

Methods of executing APL expressions using software interpreters are well known, but the possibility of execution at the hardware level presents opportunities for developing new methods. The outstanding example is a machine architecture designed by P. S. Abrams.[2] Abrams' design was purely theoretical; he made no attempt to build the machine. However, Schroeder and Vaughn[3] describe hardware intended to implement the Abrams design.

APL systems

In all APL systems,[4] processing takes place in a piece of memory called a workspace, which contains APL programs, data, and control information, as well as some unused space. Each user can have a library of many workspaces, but only one can be active at a time. When APL execution begins, the user is provided with a clear workspace—one containing some control information but no programs or data. The user can enter programs into the clear workspace, or he can replace it by loading a workspace from his library. When the system is ready for input, the user can type an APL command, he can define or edit an APL function, or he can type an APL statement. Statements, which are executed immediately, may initiate extensive calculations by calling previously defined APL functions.

The major components of an APL system are as follows:

- A supervisor communicates with the terminal, loads workspaces, services interrupts, and so on.
- A translator accepts APL statements or functions, translates them into internal form, and stores them in the active workspace. (The external form of an APL statement or function is not saved. A translation is not compilation, but simple one-for-one mapping; it is analogous to the assembly process, which converts assembler language instructions into binary machine language instructions. The system saves a table of

user defined names so that the translator can easily translate from internal to external form when a display of a function is required.)

- An interpreter carries out the interpretive execution of APL statements.
- One or more auxiliary processors communicate with the operating system, disk files, input/output devices, and other processors. A shared-memory processor provides common storage for the auxiliary processors.[4, 5, 6]

historical background

The problems and benefits of implementing APL in a microprogram were investigated in 1973 by the present authors and J. W. Lageschulte.[7] The work was done on a System/360 Model 25. It provided valuable experience, and the implementation functioned correctly, but the machine was too small and too slow to support a useful multi-user system. This work and the work of Grant, Greenberg, and Redell,[8] for example, were attempts to build a machine that would execute APL only.

On completing the Model 25 work, we set out to provide for direct execution of APL on a multi-user machine that could support a virtual-memory operating system. We decided it would be unreasonable to insist that the entire machine be dedicated to APL. The result was a plan to supplement the microprogram on a System/370 Model 145 so that the machine could directly execute both System/370 instructions and APL statements. Implementation of this plan led to development of the APL Assist microprogram,[9] which was made available to IBM customers in September 1974.

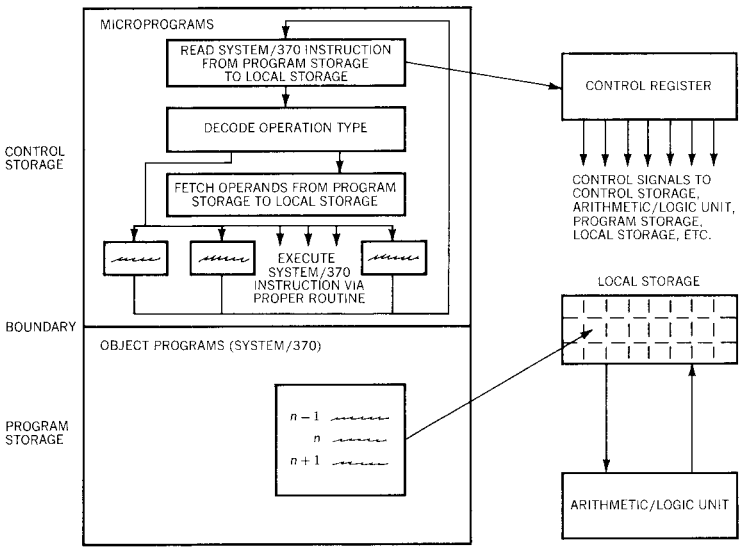
In this paper we describe what the APL Assist does, we discuss how it works and the way it interacts with System/370 software, and we compare execution times for APL programs with and without the Assist.

Microprograms

Some of the component parts of a System/370 central processing unit are shown in Figure 1.[10] The local storage contains general purpose registers, floating point registers, and some working registers for use by the microprograms. The control storage contains the microprograms for executing System/370 instructions.

In many microprogrammed machines, the control and program storages are physically separate and are implemented in different technologies. Figure 1, however, is based on the Model 145, which has a single storage divided into two parts. The boundary can be moved, under microprogram control, to give a maximum control storage of 64K bytes.

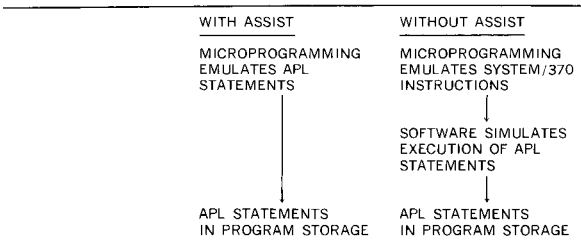
Figure 1 General microprogram flow



On a Model 145, microinstructions can perform simple operations such as register-to-register or register-to/from-storage moves, 32-bit integer addition or subtraction, and 8-bit logical functions. The hardware reads a System/370 instruction from program storage, determines the operand types, fetches the operands if appropriate, and branches to the appropriate microprogram routine, which performs the sequence of elementary steps that produce the results required by the instruction. This hardware assisted simulation of a machine is referred to as emulation—in this case, System/370 emulation. Some instructions require only a few microprogram steps. For example, CR (compare register) takes five steps requiring about .2 microsecond each; total execution time, including the decoding time, is 1.578 microseconds. Other instructions take many steps; for example, M (multiply) takes about 80 steps and has an execution time of 20.077 microseconds.

Installation of the APL Assist can be illustrated by the effect it would have on the system represented by Figure 1. The boundary would have to be moved down by approximately 20000 bytes, one new microroutine would have to be added to handle a new System/370 instruction called APLEC (APL Emulator Call), and 50 new microroutines would have to be added to perform the APL emulation. When the APLEC instruction is used, the machine switches from System/370 emulation to APL emulation; when the instruction has finished, the machine switches back to System/370 emulation. The fact that the control and program storages are separated by a movable boundary is not essential to the APL Assist, but it does facilitate its installation.

Figure 2 APL execution with and without the APL Assist



APL can be executed with or without the APL Assist. The difference between the two modes of execution is illustrated in Figure 2.

The workspace

An APL workspace is divided into five sections: the address table, the control words area, the stack, the data area, and the system area. The use of the address table is discussed below. The control words area contains status information such as the name of the current function, the address of the next APL token to be scanned, and the address of the top of the stack. The stack holds temporary results during the execution of a statement. It also holds the name of the calling function and the values of local variables in all function calls. The data area contains the values of APL variables and functions. The system area contains information used by the translator and the system but not required by the emulator; for example, the emulator never needs to know external names.

Each external (user defined) name, such as *ALPHA*, is associated with an internal name. An internal name is simply a number that is a multiple of four. For example, if numbers 0 through 96 were reserved for the system, the translator would allocate 100, 104, and so on as it encountered each new external name. A name such as *ALPHA* could be used in several different ways in a workspace; for example, it might be the name of a global function and also the name of a local variable. Each occurrence of *ALPHA* would be translated into the same internal name, the meaning of which would be determined by its context.

If an item such as *ALPHA* has an internal name n, then the word at offset n from the base of the workspace is the address-table entry for *ALPHA*. This word has one of the following forms:

Immediate form	S vipu D V
Address form	S vipu A

in which S is 4 bits long, D is 8 bits, V is 16 bits, A is 24 bits, and v, i, p, and u are one bit each. S gives the syntactic class, showing whether *ALPHA* is a variable, a niladic function, a monadic function, or a dyadic function. Bit v shows whether *ALPHA* has a value or not. Bit i distinguishes between the immediate and address forms. Bit p has a value of 1 if *ALPHA* is a permanent object, as opposed to a temporary intermediate result. Bit u is unused. The immediate form is used if *ALPHA* is a scalar character, a logical argument (0 or 1), or a small integer. In the immediate form, V gives the value of *ALPHA*, and D gives the data type (character, logical, or integer). The address form is used for all other cases. A is the address of the value block, which is stored in the data area. The value block contains a descriptor that shows whether the varaible represents a logical argument, an integer, a real value, or a character string, and whether it is a scalar, a vector, or an array. The value block also contains the element count, the rank, the length of each dimension, and the value of each element.

The formats for variables and functions used by APL/CMS[5, 9] are quite different from those used in our earlier work,[7] although the APL objects represented are used in much the same way. Some of the differences are as follows: Logical values are stored 8 bits per byte. The address table is 32 bits wide, rather than 16 bits as in the Model 25. Information about the syntactic class is stored in the address table, thereby avoiding an extra memory fetch during a statement scan. When possible, the value is stored directly in the address table. Addresses are 24 bits long, and the emulator uses virtual addressing so that workspaces as large as 16 megabytes can be used. The maximum rank of any array is 63. The maximum number of elements in any vector or array is 16 777 215. There are some additional data types, which are described in a later section of this paper.

Using the APL emulator

Let WORKBASE denote a specific System/370 general purpose register. Let NEXT denote a specific word in the control words area of the workspace. APL/CMS puts the address of the beginning of an APL statement into NEXT, it loads the address of the base of the workspace into WORKBASE, and then it uses the APLEC instruction, which causes the System/370 microprogram to give control to the microprogram of the APL emulator. The APL microprogram gets the address from NEXT and starts executing the APL statement. It continues executing (possibly with interrupts—see below) until it reaches the end of APL execution. The APL microprogram then sets the System/370 condition code, possibly sets a return code in a register, and returns control to the System/370 microprogram. System/370 execution

then resumes with the instruction following APLEC. A condition code of zero denotes a normal end of APL execution. A nonzero condition code denotes an abnormal end or an interrupt of APL execution, and a register shows the reason for the termination of APLEC.

examples of APL execution

Assume that a user types the following function to find the variance of a vector of numbers:

```
∇   RESULT ← VARIANCE X;N
[1] N ← ⍴X
[2] RESULT ← (N×+/X*2)-(+/X)*2
[3] RESULT ← RESULT÷N×N-1
∇
```

The system translates the function into internal form and stores it in the workspace. Suppose the user now types

```
ALPHA ← 4 3 7 11 2
```

The system translates this statement into internal form, stores it in the workspace, sets NEXT and WORKBASE, and calls the emulator. The emulator assigns the vector to *ALPHA* and returns with a zero condition code. Now the user types

```
BETA ← VARIANCE ALPHA
```

The system translates this statement into internal form, stores it in the workspace, sets NEXT and WORKBASE, and calls the emulator. Table 1 shows the relationship that exists at this stage between internal and external names. Now assume that a, b, c . . . denote addresses in the data area, and that the control word NEXT contains c. The address table and the data area, then, contain the information indicated in Table 2.

To give a specific example, the following hexadecimal numbers are stored beginning at location c:

0074 0068 7001 0078 A001 0003

These numbers, respectively, are the internal name of *ALPHA* (116 decimal is 74 hexadecimal), the internal name of *VARIANCE*, the code for left arrow (this code and all others differ from the codes used in our earlier work[7]), the internal name of *BETA*, an end-of-statement marker, and an end-of-execution marker.

When the APLEC instruction is issued, the APL emulator gets the half-word from location c (that is, the half-word 0074), finds it to be the name of a variable, and puts it on the stack. The emu-

Table 1 Relationships between internal and external names

INTERNAL		EXTERNAL
Decimal	*Hexadecimal*	
100	0064	*RESULT*
104	0068	*VARIANCE*
108	006C	*X*
112	0070	*N*
116	0074	*ALPHA*
120	0078	*BETA*

Table 2 Contents of address table and data area

ADDRESS TABLE

Location		*Contents*
Decimal	*Hexadecimal*	
100	0064	variable, no value
104	0068	monadic function, in data area at location a
108	006C	variable, no value
112	0070	variable, no value
116	0074	variable, value in data area at location b
120	0078	variable, no value

DATA AREA

Location	*Contents*
a	internal text for the function *VARIANCE*
b	an integer vector with elements 4 3 7 11 2
c	internal text for the statement *BETA* ← *VAIRANCE ALPHA*

lator then gets the half-word from location c + 2, finds it to be the name of a monadic function, and calls the function. The mechanism of scanning the statement, calling the function, etc., is similar to that reported earlier.[7] Next, the emulator executes the statements of *VARIANCE*, returns the result, assigns the value to *BETA*, sees the end-of-statement marker, and, upon encountering the 0003, sets the condition code to zero and switches back to System/370 mode.

At this stage, NEXT contains c + 12, and the address-table entry for *BETA* shows it to be a variable with a value. If the value were an integer, it might be stored in the address table itself; in this case, the value is a floating point number, so the address-table entry contains the address of a value block in the data

area. To summarize, the APL emulator gets control when APLEC is used, it carries out the calculation specified by the statements and functions in the workspace, and finally it returns control to the APL/CMS software.

As another example, if the user types

```
VARIANCE ALPHA
```

the internal hexadecimal code is simply

0074 0068 A001 0003

The emulator calls *VARIANCE*, calculates the result in the usual way, returns from *VARIANCE*, and puts the result on the stack. The combination of a result on the stack and the end-of-statement marker causes the emulator to set the condition code to nonzero, set the return code to *print*, and return to System/370 mode. APL/CMS detects the nonzero condition code and discovers that printing is required. The item to be printed is indicated by one word on the stack. This word has the same format as a word in the address table. It contains either the value or the address of the value to be printed. The system formats and prints the item, then issues another APLEC. NEXT now contains the value c + 6, which it had on the *print* exit. The emulator reads the next half-word, finds that it is the end-of-execution marker, and makes a normal exit.

To consider one more example, if the user types

```
BETA ← VARIANCE 'PQRS'
```

the emulator executes in the normal way until it gets to the +/*X* in the second statement of *VARIANCE*. At that point it detects the fact that *X* is a character variable and that +/ is therefore undefined. The emulator sets the condition code to nonzero, sets the return code to *domain error*, and returns to System/370 mode. At this stage, NEXT points to the item following the +. Remember that the statement is stored in reverse order—the emulator has scanned

```
2 * ) X / +
```

and NEXT is now pointing to the (. The system can reverse translate so as to display the statement in error, and it can use NEXT to point to the token at which the error was detected. At this stage, the changes in the address table are:

108	variable, value in data area at location d
112	variable, value in this word, value is integer 4

Location d in the data area begins a four-element vector, '*PQRS*'. The user can ask for the values of *X* or *N* to be displayed. He can also resume execution in the normal APL manner.

Interrupts and page faults

The time required for the execution of the APLEC instruction depends on the APL functions specified in the workspace. Since the operating system is multiprogramming many tasks, it is imperative for APLEC to be interruptable. There is a trigger in the hardware which is set when an interrupt is pending. The emulator tests this trigger at frequent internals, and when it is set, the emulator stores status information in the workspace and allows a normal System/370 interrupt to occur. The interrupt process is not detectable by the program that issued the APLEC.

The APL emulator handles page faults in a similar way. When a page fault occurs, the emulator stores status information in the workspace and allows the fault to pass to the operating system in the normal way. Consider the case in which the emulator is evaluating *A+B*, where *A* and *B* are integer vectors of 20000 elements each. The result will also contain 20000 elements. It would be unreasonable for the emulator to require that 60 pages (20 pages of 4096 bytes each for *A*, 20 pages for *B*, and 20 pages for the result) be in real memory simultaneously. The emulator requires that at any one time, real memory contain only the first page of the workspace and one page of *A* or *B* or the result. To avoid an excessive number of page faults, however, the first workspace page and one page each of *A*, *B*, and the result should be in memory simultaneously.

The APL Assist does not modify the behavior of the address-checking or protection mechanism. If, for example, an address-table entry is in error and the APL Assist attempts to store data in a protected area, then a System/370 protection exception occurs.

File input/output

The design of APL[6]—with a main processor to execute APL statements, and one or more auxiliary processors for file input and output—is readily adaptable to our situation if we consider the emulator to be the main processor, and if we provide System/370 routines to act as auxiliary processors. The APL system has to provide a facility that allows the main processor to share an APL variable with an auxiliary processor. The following statements illustrate the use of APL/CMS[5] auxiliary processor 110 to write the APL variables *A*, *B*, and *C* as three separate records in a file called TEST DATA:

```
X←'TEST DATA'
110 □SVO 'X'
X←A
X←B
X←C
```

□*SVO* is an APL system function; *SVO* stands for *shared variable offer*. The emulator assigns the character string '*TEST DATA*' into *X*. It begins executing the next statement in the normal way, but discovers that □*SVO* is a system function, sets a register to indicate this, and exits. A System/370 program is used to pass the value of *X* to the auxiliary processor, which uses the value to name a CMS file. The APL/CMS system records the fact that *X* is a shared variable by changing the syntactic class of the address-table entry of *X* from *variable* to *shared variable*. The system then issues another APLEC to resume APL execution.

Execution now proceeds in the normal way until the emulator is required to assign a value into *X*. Assigning a value into a shared variable means that the new value must be stored in the workspace and must also be sent to the auxiliary processor. The emulator sets a register with the name of *X*, sets another register with an indication that *assign into shared variable* is required, and exits. The system passes the value of *X* to the auxiliary processor and issues an APLEC to resume APL execution.

Completeness

The APL language has a large number of primitive functions.[4] In other languages, functions such as matrix inversion and output formatting are library routines, but in APL they are part of the primitive function set. In any practical emulator there must be limitations on the size of the control storage, and it may be impossible to emulate the whole of APL in the available space. Even without a strict limit, we would have to recognize that the control storage is a valuable resource and must not be used wastefully.

Methods of implementing APL in a small amount of storage include restricting the number of data types and using simple, general purpose algorithms. Both of these methods would have a disastrous effect on performance. Our approach is to use many data types, to strive for optimal efficiency in frequently used functions, and to omit any function that would not benefit from the use of microprogramming. The emulator provides a mechanism for calling software routines to compute the value of functions not provided in microprogramming. Most of these routines are written in System/370 machine language, but a few are written in APL. If the emulator requests the system to carry out

some function, the system can return either the name of the result or the name of an APL function that can compute the result.

The emulator performs all the operations required for statement scanning and syntactic analysis, for calling and returning from functions, for getting and freeing blocks in the data area, and for subscripting and the branching and assignment statements. It performs most of the scalar operations on scalars, vectors, and arrays. It performs all of the following functions: size, reshape, ravel, catenate, laminate, compress, expand, index generator, index of, membership, and reverse. It handles most uses of the transpose function and some uses of the take, drop, rotate, reduce, and inner product and outer product, but for other uses it calls on the software.

The emulator calls on the software for the translation part of execute and all uses of encode, decode, grade, scan, format, deal, and matrix invert. There would have been no difficulty in writing microinstructions for these functions had we chosen to do so. For example, subscripting is handled by microprogramming even though it is considerably more complicated than the grade function, which we chose to implement in software.

Our decision to exclude any operation from the emulator was based on a number of considerations. If an operation requires a large amount of floating point arithmetic, it will not be executed significantly faster by microprogramming. For this reason, matrix inversions are handled completely by software in our system. Similarly, logarithms are computed by software, but the initial analysis, successive operand fetching, and storing of the result are done by a microprogram. In certain operations, such as computing an inner product and encoding, the key to rapid execution lies in recognizing many special cases and then selecting an algorithm that is especially suited to the case at hand. In computing an inner product, there is not room in control storage to optimize all cases, so certain common cases (in which one argument is scalar or both arguments are vector) are handled by microprograms, and software is used for the other cases.

Some APL operations require conversion between external and internal form, and it is natural to exclude these operations from the emulator. For example,

$A \leftarrow B + \blacktriangleleft C$

requires the system to treat the character string C as an APL expression in external form, convert it to internal form, execute the internal form, add the result to B, and place the sum in A. The emulator handles this operation by getting C, ascertaining that it has a value, and passing this value to a software routine.

The software routine converts the value to internal form and returns it to the emulator. The emulator executes the internal form, adds the result to *B*, and stores the sum in *A*.

Design considerations

In our system the layout of the workspace, the format of objects in the workspace, and the encoding of operators and descriptors were chosen to maximize execution efficiency and, where possible, to minimize the number of microinstructions. Many of the decisions involved did not depend on the fact that we were using microprogramming. For example, the storing of scalars in the address table, rather than in the data area, improves execution speed and reduces the possibility of page faulting in both microprogram and software implementations.

The major aspects of microprogramming that influenced our design were that a test and branch on one or two bits is very fast and fetching words from memory is relatively slow. In evaluating an expression such as *A*α*B* (in which α is any APL primitive function), the system must make tests such as "is α scalar or mixed" and "is α a logical, comparision, or arithmetic operation."

In a software implementation, one probably would encode α so that the tests could be phrased "is α less than n1" and "is α less than n2 or greater than n3" (where n1, n2, and n3 are integers). This method would require two microinstructions for the first test and four for the second. We chose the internal representation for α so that the tests could be phrased "branch according as α bit 4 is 0 or 1" and "branch according as α bits 5, 6 are 00, 01, 10 or 11." These two tests take one microinstruction each.

A descriptor is associated with every variable that has a value. The descriptor is either in the address table or in the first word of the value block. If DA and DB denote the descriptors of *A* and *B*, then DA and DB are stored in a register at an early stage in the evaluation of *A*α*B*. The system must make the test "is *A* a scalar or a one-element vector." In a software implementation, the test might be phrased "if the element count of *A* is 1 and the rank is less than 2, then the answer is yes." This test would require two memory fetches. We chose the representation of DA so that bits 1, 2, and 3 would identify *A* as follows:

bit 1	2	3	
0	0	0	*A* is a scalar
0	0	1	*A* is a one-element vector
0	1	1	*A* is a one-element array
1	0	1	*A* is a vector (zero or > 1 elements)
1	1	1	*A* is an array (zero or > 1 elements)

The test "is A a scalar or a one-element vector" can now be phrased in the single microinstruction "are DA bits 1, 2 equal to 00." If DC is the logical "or" of DA and DB, the test "are DA and DB both scalar" becomes "is DC bit 3 equal to zero."

On the System/370 Model 145, the APL emulator uses four times the amount of control storage used in the System/360 Model 25 implementation. Much of this additional storage is used in implementing many more functions in microprogramming, but some is used for more sophisticated algorithms. As an example, the internal representation uses a data structure called the APV (arithmetic progression vector). This structure has a special descriptor and three data words: $V1$, $V2$, E. The APV is used to represent the integer vector with elements $V1$, $V1+V2$, $V1+2\times V2$ . . . $V1+(E-1)\times V2$.

The emulator uses the APV form in the following way: The APL expression ωJ stands for the vector $1,2,3 \ldots J$. If the emulator encounters the expression ωJ, it produces an APV if J is greater than 1. The APV will have $V1=1$, $V2=1$, $E=J$. Many of the emulator routines check for APV arguments and, where possible, produce an APV result. For example if an APV is multipled by a scalar X, the result is formed by multiplying $V1$ and $V2$ by X. If an APV is to be reversed, then $V1$ is replaced by $V1+(E-1)\times V2$, and the sign of $V2$ is changed.

One advantage of the APV is that it saves $E-2$ words of memory. It also saves time. For example, multiplication by a scalar requires three rather than E multiplications. (Two multiplications are mentioned above, but the system also forms $(V1\times X)+(V2\times X)\times(E-1)$ to check for integer overflow.) The major reason for implementing APVs is that they occur in many subscript expressions.

All subscript evaluation is done by a microprogram routine using the methods we have described elsewhere.[11] Subscript evaluation is complicated by the fact that arrays of ranks 1 through 63 can be subscripted. Each subscript can be a scalar, a vector, or an array. To improve execution efficiency, it is important to recognize special cases dynamically. The time required to execute $U[V]$, for example, is a constant plus some factor times the number of elements in V. The constant is almost independent of the characteristics of U and V, but the factor is strongly dependent on U and V. The emulator checks the form of V. If V is an APV and its $V2$ part is unity, then the subscripted elements can be moved in a single block. The gain in speed is particularly significant if U is a logical vector; it takes fewer microprogram instructions to move 32 bits aligned on a word boundary than to move a single bit. In many cases, array subscripts can be analyzed and reduced to simpler and more efficient forms.[11]

APL uses *call by value* when a defined function is invoked. Consider the statement.

```
BETA ← VARIANCE ALPHA
```

in which *VARIANCE* has the header

```
∇RESULT ← VARIANCE X;N
```

The usual method of passing the value of *ALPHA* to *X* is to make a copy of *ALPHA* and attach it to the address-table entry for *X*. This method is simple and effective, but it is very inefficient if *ALPHA* is a long vector.

A second method is to pass the address, instead of the value, of *ALPHA*. This method appears to be simple, but it is not. Blocks in the data area are assigned dynamically, and when the data area is full, a garbage collection is done. A garbage collection may change any or all addresses. There is another problem in passing the address: the value of *X* may be changed, and this must not change the value of *ALPHA*. Similarly the value of *ALPHA* may change (since *ALPHA* is global in *VARIANCE*), and this must not affect the value of *X*.

The solution used in the APL emulator is to define an object called a synonym block. This block has a descriptor showing it to be a synonym, and it has four half-words denoted here by N, T, L, and R, which are internal names. This synonym block states that N (that is, the variable with internal name N) is synonymous with L and R, and that the value of N, L, and R can be found in the block with name T. If N is synonymous with only one other variable, either L or R is set to −1. If N is synonymous with more than two other variables, L and R form a chain for passing from one synonym block to another.

In the example above, assume that *X* and *ALPHA* have internal names 108 and 116, that T has internal name 200, and that a, b, and c denote addresses in the data area. Before the function call, the address table contains

108 variable, no value
116 variable, value in the data area at location a

and the data area contains

location a value block for variable 116.

After the function call, the address table contains

108 variable, value in the data area at location b

116 variable, value in the data area at location c
200 variable, value in the data area at location a

and the data area contains

location a value block for variable 200
location b synonym block 108 200 116 −1
location c synonym block 116 200 −1 108

Note that these blocks (and all other blocks in the data area) contain no addresses, so garbage collection is a simple and rapid operation. The internal name T (the number 200 in the example) does not have an external name; it is only used internally by the emulator. If an attempt is made to change the value of *ALPHA*, the emulator will free the old value of *ALPHA*. The *free* routine will free the synonym block for *ALPHA* and it will detect that *ALPHA* was synonymous with *X*, then it will look at the synonym block for *X* and discover that there are no more synonyms, so it will free the synonym block for *X* and the name of T and cause the address-table entry for *X* to point directly to the value block. The APL emulator uses synonyms in calling functions and also in operations such as raveling an array, and in some cases of assignment.

Performance

We measured the performance of the APL emulator by measuring the CPU times required to execute 19 test problems, with and without the emulator, on a dedicated machine. The problems covered a wide range of applications in numerical analysis, statistics, linear programming, text processing, and compilation. We compared three implementations of APL:

(A) APL/CMS running on a System/370 Model 145 with the emulator installed;

(B) APL\360 running on the Model 145;

(C) APL/CMS running on the Model 145 using a software interpreter instead of the emulator.

The test problems ran 2 to 20 times faster on system A than on B. That is, one problem ran twice as fast and one ran 20 times as fast, and the running times of the other problems were between these extremes. Similarly, the test problems ran 1.5 to 2.5 times faster on C than on B. These figures were obtained with one user on the system. If there were many interactive users, the system's behavior would be controlled largely by operating-system functions such as terminal handling, multiprogramming, and virtual-memory paging. The average user would see little differ-

ence between systems A and C, but a user executing a long APL calculation on system A would, in most cases, see a dramatic reduction in CPU time.

The performance figures indicate that, in a controlled environment (that is, with a single user on a dedicated system), CPU time was reduced by an average factor of 2 because of new implementation techniques (new method of syntactic analysis, use of synonyms, APVs, etc.) and by an average factor of 5 because of the use of microprogramming. However, there were wide variations in the amount of CPU-time reduction attributable to microprogramming. The reason for these variations is discussed below.

The execution speed of particular statements is not necessarily a good indicator of overall performance. The execution speed of particular statements can be used, however, to illustrate some of the factors that determine the relative performance of microprogram and software execution of APL. We measured the execution time required by 13 specific APL statements, using APL/CMS with and without the APL emulator installed. Table 3 shows the results in terms of the ratio of execution time *without* the emulator to execution time *with* the emulator. *S*, *V*, and *A* denote *scalar*, *vector*, and *array*, respectively, and *L*, *I*, *R*, and *C* denote *logical*, *integer*, *real*, and *character*. Trailing numbers denote dimensions. For example, *IV*100 represents an integer vector with 100 elements, and *RA*10_20 represents a 10-by-20 real array. The last example in the table (the transposition of a 10-dimensional array) is intended to show that the emulator does not lose its power even in very complicated operations.

The execution times were measured on a system running many jobs, so errors of as much as 10 percent can be expected. This margin of error, however, is quite accurate enough for our purpose. The times were measured as follows:

```
     M1←M2←M
     T1←⎕AI
L1:  →(0<M1←M1-1)/L1
     T2←⎕AI
L2:  X←IS+IS
     →(0<M2←M2-1)/L2
     T3←⎕AI
     TIME←((T3-T2)-(T2-T1))[2]÷M
```

The value of *M* was made large enough so that it would average the perturbations caused by input/output, page faults, and so on.

One way to check this timing method is as follows: The inner loop used by the software in evaluating the statement

Table 3 Execution times of specific APL statements in terms of the ratio of time required without the APL emulator to time required with the emulator installed

STATEMENT	*RATIO*
`X ← IS+IS`	12.56
`X ← IV100+IV100`	3.07
`X ← IA5_20+IA5_20`	3.23
`X ← IV1000+IV1000`	2.30
`X ← RS+RS`	8.42
`X ← RV100+RV100`	1.76
`X ← RS÷RS`	4.80
`X ← RV100÷RV100`	1.05
`X ← RA10_10⌹RA10_10`	1.06
`X ← IV200[IS]`	14.54
`X ← IV200[IV100]`	3.38
`X ← IV200[⍳100]`	6.54
`X ← ,⍉(10⍴2)⍴IV1024`	2.67

`X ←+/IVN`

consists of two instructions (indexed A and BXH) having an execution time of 5.977 microseconds.[12] We used the method described above to time the execution of this statement with two values of N: N = 1 and N = 1001. The difference between the times was 6.098 milliseconds, an error of 2 percent.

Statements like those in Table 3 have the general form

$A \leftarrow B \ \alpha \ C$

in which α denotes any APL primitive function. The time required to execute such statements is typically indicated by an equation of the form $P + Qn$, in which n is the number of elements in the result (except for functions like matrix divide and grade), P is the time required to analyze the statement and the operands and get space to store the result, and Q is the time required to compute one element of the result.

The value of P when microprogram execution is used is typically ten times smaller than the time required when software is used. The value of Q does not differ so greatly, and in many cases the times for microprogram and software execution are almost the same. It follows that if the number of elements is small, then microprogram execution is significantly faster than software. If n is 10 or 20, the relative performance depends on the relative sizes of P and Q. In the examples shown in Table 3, P is much larger than Q for integer addition, but about the same for real division. If n is large, microprogram and software execution times usually are quite similar.

Some software routines may use a special algorithm which makes them faster than the corresponding microprograms. For example, if *LVZ*1000 is a logical vector with 1000 zero elements, and *LVO*1000 is a vector with 1000 ones, then the following execution-time ratios are obtained:

STATEMENT	RATIO
X←+/*LVZ*1000	0.30
X←+/*LVO*1000	1.17

The microprogram time depends on the number of elements, whereas the software time depends primarily on the number of ones.

examples

Two examples serve to illustrate how these results apply to complete programs. An FFT (fast Fourier transform) calculation requires many floating point multiplications. It would be expected that microprogram and software execution would require comparable times for an FFT on a long vector. For an FFT on eight complex points, we have found microprogram execution to be 4.6 times faster than software; but for 1024 complex points, it is only 1.5 times faster. As a second example, the APLGOL compiler,[13] which is written in APL and uses no floating point operations, has to break APLGOL statements into tokens which are short vectors. It would be expected that microprogram execution would be significantly faster than software. Our test case was to compile the APLGOL compiler itself; in this case, the microprogram execution was 7.33 times faster than the software.

Summary

The architecture of an APL machine is radically different from the architecture of a machine like the IBM System/370. The APL Assist demonstrates that both architectures can be supported on one central processing unit. It also shows how an APL emulator can function in the environment provided by the VM and VS operating systems without demanding any special privileges.

In many high-level languages, a compromise has been made between making the language easy to use and enabling it to be compiled. In APL this compromise has not been made. APL is designed to be both powerful and easy to use; consequently it cannot be compiled into conventional machine language instructions. The APL Assist shows that the machine can be designed to fit the language. Making the machine fit the language does not add anything to the language, but, as we have shown, it can result in an impressive improvement in execution speed.

ACKNOWLEDGMENTS

We are grateful to many persons within IBM, particularly those who answered our many questions about the System/370 Model 145 microprogram and those who used and made comments on successive versions of the APL emulator. This study would not have been possible without the system software provided by M. J. Beniston. Additional aid in developing and testing APL/CMS was provided by J. W. Lageschulte and others. We are particularly grateful to R. J. Creasy for advice and encouragement during all phases of the study.

CITED REFERENCES AND FOOTNOTES

1. J. E. Nicholls, *The Structure and Design of Programming Languages*, The Systems Programming Series, Addison-Wesley Publishing Co., Reading, Massachusetts (1975).
2. P. S. Abrams, *An APL Machine*, Ph.D. thesis, Stanford University, Stanford, California (1970); available as Document AD-706 741 from the National Technical Information Service, United States Department of Commerce.
3. S. C. Schroeder and L. E. Vaughn, "A high order language optimal execution processor," *Proceedings, ACM-IEEE Symposium on High-Level-Language Computer Architecture*, University of Maryland, November 1973, pp. 109–116.
4. *APL Language*, order number GC26-3847, IBM Systems Library, General Products Division, Programming Publishing Department, 1501 California Avenue, Palo Alto, California 94304 (1976).
5. *APL/CMS User's Manual, Programming RPQ MF2608*, order number SC20-1846, IBM Scientific Center, APL/CMS Publications, 1530 Page Mill Road, Palo Alto, California 94304 (1975).
6. R. H. Lathwell, "System Formulation and APL Shared Variables," *IBM Journal of Research and Development* **17,** No. 4, 353–359 (1973).
7. A. Hassitt, J. W. Lageschulte, and L. E. Lyon, "Implementation of a high level language machine," *Communications of the ACM* **16,** No. 4, 199–212 (1973).
8. C. A. Grant, M. L. Greenberg, and D. D. Redell, "A computer system providing microcoded APL," *Proceedings, 6th APL Users Conference*, Anaheim, California, May 1974, pp. 173–179.
9. The APL Assist was first made available as RPQ S00256 for the System/370 Model 145. The RPQ is used by APL/CMS (IBM Programming RPQ MF2608, program number 5799-ALK), which runs under the CMS component of VM/370. Subsequently, an APL Assist Feature (number 1005) was developed for operation with VS APL(IBM program product 5748-AP1); VS APL runs under both CMS and the VSPC program products (5746-XR3 for DOS/VS, 5740-XR5 for OS/VS1, and 5740-XR6 for OS/VS2). In this paper, the term *APL Assist* refers to the RPQ—although for the most part, the discussion applies to both because the RPQ and the Feature are almost identical on the Model 145. The Assist Feature is available on System/370 Models 135, 138, 145, and 148.
10. Figure 1 is reproduced from *Introduction to Microprogramming*, out of print, IBM Corporation (1971).
11. A. Hassitt and L. E. Lyon, "Efficient Evaluation of Array Subscripts of Arrays," *IBM Journal of Research and Development* **16,** No. 1, 45–47 (1972).
12. *IBM System/370 Model 145 Functional Characteristics*, order number GA24-3557, IBM Systems Library, System Products Division, Department K10, P.O. Box 6, Endicott, New York 13760 (1975).
13. R. A. Kelley, "APLGOL, An Experimental Structured Programming Language," *IBM Journal of Research and Development* **17,** No. 1, 69–73 (1973).

GENERAL REFERENCES

Y. Chu (editor), *High-level Language Computer Architecture*, Academic Press, New York (1975). Describes a number of machines designed to directly execute high-level languages.

P. M. Davies, "Readings in microprogramming," *IBM Systems Journal* **11,** No. 1, 16–40 (1972). Provides an introduction to microprogramming and a survey of the literature through 1971.

C. W. Gear, *Computer Organization and Programming*, McGraw-Hill Book Company, New York (1969). Contains a chapter on machine design and microprogramming.

A. B. Salisbury, *Microprogrammable Computer Architectures*, Elsevier Press, New York (1975). Describes the microprogramming of several different machines.

Reprint Order No. G321-5041.