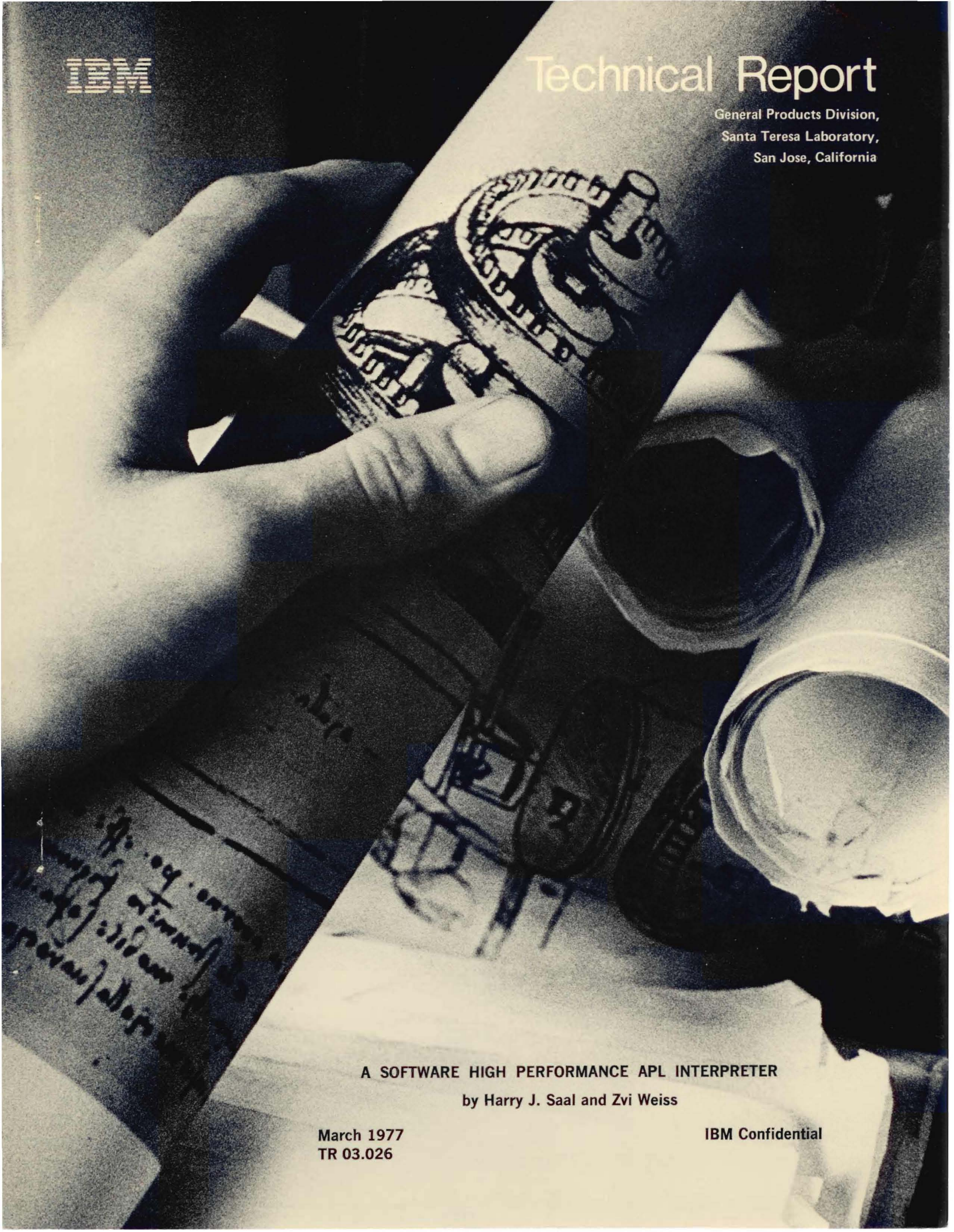


IBM

Technical Report

General Products Division,
Santa Teresa Laboratory,
San Jose, California

A SOFTWARE HIGH PERFORMANCE APL INTERPRETER

by Harry J. Saal and Zvi Weiss

March 1977
TR 03.026

IBM Confidential

IBM Confidential
March 1977

TR 03.026

A SOFTWARE HIGH PERFORMANCE APL INTERPRETER

by

Harry J. Saal and Zvi Weiss*

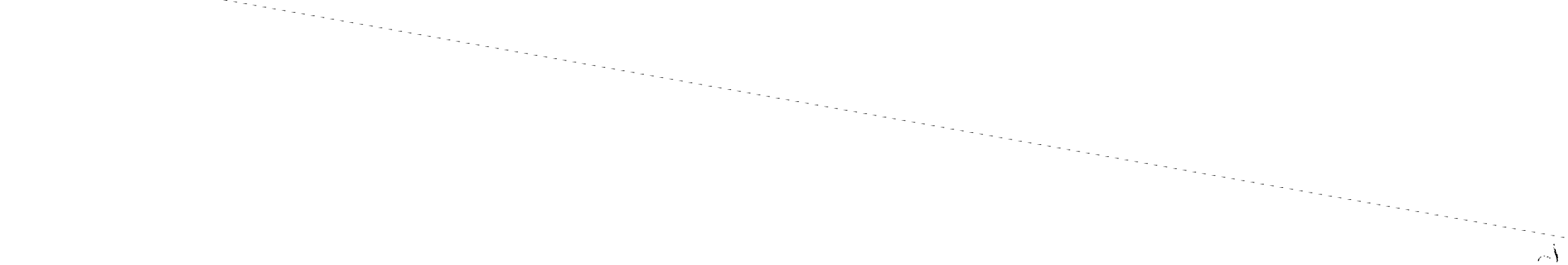
*IBM Israel Scientific Center, Haifa, Israel

International Business Machines Corporation

General Products Division

Santa Teresa Laboratory

San Jose, California



ABSTRACT

The design of a high performance APL system is presented along with an evaluation of the performance improvement measured on a partial implementation. The system contains a compiler which translates APL into the instructions of a virtual APL machine. Numerous special techniques suitable for optimized interpretation of this virtual machine entirely in software on a System 370 are described. The overhead for executing APL programs has been reduced by a factor ranging between 5 and 10 when compared to conventional interpretive systems. One realistic example is analyzed in depth; there the compiled version runs 6 to 8 times faster than APLSV (Version 1.2).

A SOFTWARE HIGH PERFORMANCE APL INTERPRETER

by

Harry J. Saal and Zvi Weiss

INTRODUCTION

The major goal of this work was an attempt to provide, within the framework of a purely software system, high performance execution for programs written in APL. A more extensive report on this project is available from [1].

The developers of the APL Assist [2], available on the System 370 Models 135, 138, 145 and 148 have utilized the ability to add a large amount of microcode to the existing 370 instruction set to provide substantial speedup for many APL programs. This is done within the framework of an interpretive APL system, and (essentially) full compatibility with existing APL code is maintained. It appears very difficult to see how the microcode techniques used in the APL Assist can be extended to the high end of the 370 line. The same difficulty, perhaps even more seriously, applies to future systems where performance is even higher. Consequently, we have not assumed any special hardware (or microcode) beyond the standard System 370 instruction set. Later on, we will return to this point and make some concrete suggestions for achieving further substantial performance improvements but which would require only relatively minor additions to the standard instruction set.

PREVIOUS WORK

Although the subject of translating APL programs has received considerable attention in the past, little concrete success has been reported. Several investigations have centered around the embedding of APL within other high level languages, either via some automatic preprocessing [3], or by providing a library of subroutines which mimic the APL primitives [4,5]. Others concentrate on a more direct translation; in the case of [6] to ALGOL (the "machine language" of the B6700), in [7] to System 7 Assembler code, and in [8] to UC.5 microcode.

Unfortunately, all the systems mentioned above treat languages which are very far from APL as we know it. In general they demand explicit declarations for all identifiers (or else implicit ones by restricting certain primitives such as dyadic rho to literal constants for left hand arguments), don't permit varying data types (or lengths or ranks), modify the APL scope rules for variables, and don't do error checking as defined in APL. Each system suffers from different combinations of variants from this list (and many others), and none even begin to approach the objective of simply taking an existing workspace and compiling it.

DESIGN OF THE TARGET LEVEL FOR APL COMPILATION

We can get some feeling for the potential size of machine code that would result from direct translation of APL by considering other high level languages in relation to APL. One of the findings of the statistical studies on APL programs [9,10,11] was the great disparity in size (approximately 10 to 1) between typical FORTRAN and APL programs. More specific comparisons between alternate language versions of the same algorithm also bear this out. Moreover, the APL code has higher semantic complexity since, in general, it handles a wider range of possible argument types or shapes than the corresponding FORTRAN version, in addition to performing consistency and error checking, dynamic space management, etc., which are implicit in APL.

We may conclude that if one could translate APL directly to machine code for System 370, the expansion factor would be tremendous. This in turn would introduce its own complexities due to base register addressing limitations on System 370. Combined with the problem of identifying loops and side effects of APL's dynamic scoping rules and system variables, it becomes difficult to apply even simple optimizations such as constant propagation. Without strict bindings of internal representation one can't, for example, do allocation of variables to registers across loops. Thus the nature of the code generated by such a hypothetical compiler would be large, highly stylized, and unoptimized.

A natural solution to the size problem is to use a large run-time support system. One advantage is that these library routines themselves are highly optimized and remove the burden of optimization and storage from the compiler and from the compiled code. Since the generality and power of a typical APL primitive is on the average considerably higher

than that of FORTRAN or PL/I, we would expect such a system to execute almost always in the run-time library, with only small in-line bridges between subroutine calls. While this may be an acceptable solution, there are still two further problems which arise in an attempt at direct compilation.

First, if the run-time system checks for errors of length, index, etc., as it should, one would like to provide an error message facility which localizes errors as much as possible. This should be similar to current interpretive systems, and not at the level of hexadecimal dumps. In our experience, it is quite difficult to map run-time errors back to source after FORTRAN H has reordered and optimized a program. This further motivates some internal code representation which is far closer to APL than optimized System 370 machine code.

The second difficulty arises from the great difference between integer and floating point resources and instructions on System 370. Since we can't be sure if an object, even if scalar, is fixed or floating, and since something as simple as plus may on occasion cause a (legitimate) fixed point overflow, the compiled code would have to be in several alternate forms. The run-time system would have to intercept any interrupts and be able to continue computing in an alternate representation. This again inhibits optimizations and increases object code size.

Considering these difficulties, the most natural design is in fact an interpreter! We can then ask what is the role of a compiler for APL? There are indeed many aspects of APL execution that need to be left to run-time. Nonetheless, there still remain a large variety of translations and optimizations (such as syntax analysis) that need not be done repeatedly at run-time.

We therefore chose an intermediate level of representation as the compiler target language, sufficiently far from APL as to permit useful optimizations where possible, and far enough from the host System 370 as to require some form of interpretive support, i.e., a hierarchical system. This representation is the target machine for the APL compiler, and must be supported on a given host (in our case System 370). We need not be very concerned with the layout and encodings of the intermediate machine language until we choose a particular host for implementation. (For example, on 370 each field is a 32 bit word for convenience; in a microprogrammed environment one would pack fields much better than that.)

Generally virtual machine instructions are N+1 address instructions, i.e., the dyadic function *RESULT←ARG1 OP ARG2* is represented by a three address instruction of the form *(OP,ARG1,ARG2,RESULT)*. Monadic and dyadic primitive functions are represented by two and three address instructions respectively. The general cases of indexing, subscripting, and mixed output are represented by variable length instructions that hold the addresses of all the arguments.

This form of machine language is quite close to APL source, and thus can be translated back to source form along with precise error localization when required. It resembles the intermediate code generated by many optimizing compilers; however since we do not have sufficiently tight bindings on the data objects we cannot generate actual machine code from this representation.

RESTRICTIONS ON APL

The restrictions imposed on APL in the adaptive system design presented here deal with the "static appearance" of user programs in a workspace which is to be translated as a unit. By "static appearance" we mean that source statements statically convey all the required syntactic information to carry on with the different phases of the translation, thus ruling out or weakening the following features of APL:

1. *⍎X* is not supported at all.

2. *⍎EX* is limited to apply only to variables.

3. No editing of user defined functions is supported unless immediately followed by a total retranslation of the workspace.

4. ⍎ is weakened so that the character string argument can be statically parsed. In its modified version, execute is not a very useful primitive function and may be considered to be not supported. It is, however, still useful for converting numeric character vectors to numeric values, or for simple shared variable referencing (in which case the set of possible references must be known at compile time).

5. ⍞ input does not execute arbitrary APL expressions since these expressions are unknown at translation time. We restrict the input to be simple data, but which is

supplied via an interface to a remote terminal with an APL interpreter as a front end.

6. All statements must be uniquely parsible by the compiler.

The user may supply the translator with a list of names of all variables that are shared. (More precisely, they may become shared during execution at some point.) In handling non-shared variables the translator and the host machine can take some short-cuts and thus gain efficiency compared to the handling of shared variables. If the user does not wish to supply the list of shared variable names then all variables in the workspace can be assumed to be shared.

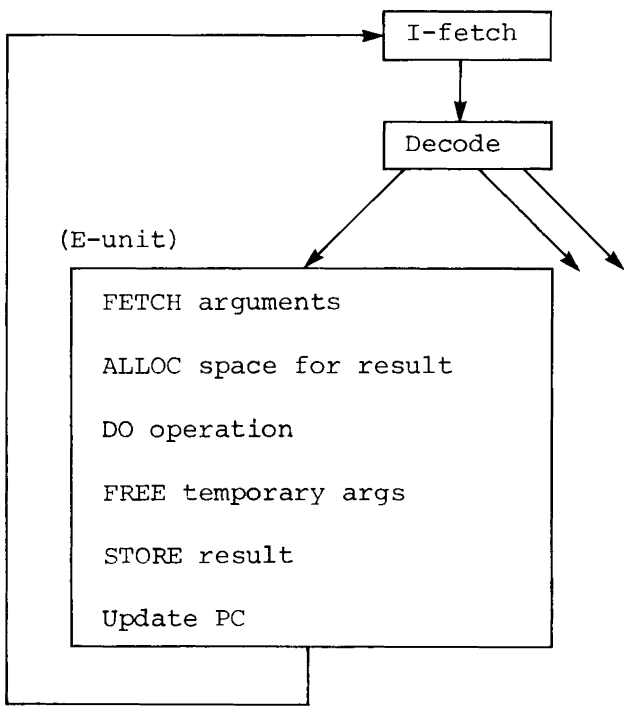


Figure 1. Phases of I-cycles

SOFTWARE INTERPRETER IMPLEMENTATION

Our software interpreter achieves speed through very extensive special case handling, wherein the decision process for determining which case applies is both complex and not necessarily very symmetrical or consistent. The cases are chosen because they are heavily used and may constitute some irregular set.

We assume that programs are well debugged using an interpretive implementation before being compiled. Thus rather than providing for rapid error traceback, or accomodation to other rare events, we take the position that execution should be as rapid as possible, leaving just enough of a trail to "pick up the pieces" whenever necessary, even if it is somewhat expensive or clumsy to do so.

Figure 1 shows the processing phases that every virtual machine instruction undergoes. Each routine FETCHes its operands, using information in the descriptor. This may involve special processing for shared or system variables which must be gotten from outside the workspace or from the system. Once the operands have been located, and any consistency requirements verified, storage can be allocated for the result. This is done again using descriptor information, to distinguish free space variables from the two temporary stacks.

After computing the result of the function, the space occupied by any temporary arguments must be returned. This must be done in reverse stack order (defined by the compiler allocation scheme). The last step is STOREing the result. This causes any necessary post processing to take place, such as transmission to the shared variable processor, or the user's terminal. For system variables the result is examined for validity and marked accordingly. If valid, system variables are then normalized in their internal representation so that any later implicit uses by execution routines do not have to worry about possible data type conversions.

Once we have recognized certain properties of operands, we may then eliminate numerous extraneous tests and operations. For example, we need not perform rank and length checking, loop setup, and computation of the amount of space needed for the result, if we are adding two integer scalars. On the other end of the spectrum, we would like to have the system execute the tightest possible inner loops when computing on

large data aggregates. We also wish to utilize the kinds of machine level optimizations that an assembly programmer would use, such as treating 32 bits at a time (in a word) or even 2048 at a time using SS instructions.

The decision to provide a particular subcase of a general APL function rests on two points: 1) the frequency with which that case is used, and 2) the ability to do significant optimizations relative to other subcases. In practice, we have been unable to use the first of these guidelines due to lack of sufficient data. Thus the second has been the more significant factor.

This tendency is also promoted by use of macros in the actual generation of code. Once a general framework has been established it is simply convenient to use it, at the possible expense of wasted space in the host interpreter for rare cases. For example, the scalar dyadic primitives fall naturally into 4 subcases, namely scalar- scalar, scalar- nonscalar, nonscalar- scalar, and nonscalar nonscalar. In addition, for the arithmetics there are integer and real versions. Consequently we have 8 subcases for each of +, -, ×, ÷, ⌈, ⌊, etc.. For relationals, there are even more. For boolean relationals we provide separate code for each of the six relationals, which is then repeated 4 times according to rank as mentioned above, i.e. 24 subcases. All the integer and real relationals are performed by an additional 8 subcases (each of which covers all the six relationals, i.e. =, ≠, >, ≥, <, ≤ without further adaptation). Hence there are in all 32 separate routines which cover all the relationals.

Undoubtedly certain of these cases are rarely used. On the other hand, the code for handling booleans is so greatly optimized (treating up to 2048 bits in a group using the minimum number of instructions) that the payoff is immense, thus the "expected value" of one of these subcases is the product of a very small probability times a very large number; this is notoriously difficult to estimate!

CODE ADAPTATION

In order to improve either the scalar domain or large aggregate domain we need to reduce the overhead of recognizing and decoding into the appropriate subcases. For scalar codes we wish to suppress redundant testing as much as possible since this can easily become the dominant cost.

We adopted the following approach, which we call "code adaptation". The virtual machine instructions known to the compiler are not, as a rule, the names of computational routines, but refer to the appropriate decoding scheme for that operation. Once the particular applicable subcase is recognized, its address is substituted in the operation code field of the virtual instruction, prior to executing the routine. As a result, on subsequent executions of the instruction, the same special case will be selected, without the decoding overhead. Since a decode might not remain valid indefinitely, we must incorporate in each special case its preconditions, i.e., the requirements on its operands for it to be applicable. This preliminary checking is done every time the operands are fetched, ensuring the consistency (and correctness) of the computation.

Every routine accesses its operands via information in the data descriptor. When fetching operands, a BALR link is made to the FETCH routine, providing a base register for the fetch routine, and a return pointer in R14. Immediately following the BALR appears the particular type of operand expected by the calling routine. This information indicates:

1) a specific expected rank (i.e., either scalar or vector or matrix etc.)
or 2) not a specific rank (only not scalar actually is used in the existing code)
or 3) don't care about rank

and

4) the expected type (bit, integer, real or character)
or 5) don't care about type.

Two further fields are specified, namely the addresses of exception handlers for cases where the rank or type requests cannot be satisfied. (A rank problem takes precedence over a type problem if both occur.)

FETCH validates the actual operands against the requirements of the calling routine. In the case of type exceptions, it performs certain conversions automatically. An empty object is always typeless, that is to say, although its actual internal representation may indicate one of the four possible types, it is acceptable as any other type. Secondly, type conversions upwards are performed automatically, from 1) bit to integer, 2) bit to real or 3) integer to real. The converted forms are created on a special internal stack, and a pointer to this area is

returned. The calling routines are totally unaware of this automatic conversion (although it is detectable in the rare case that this is necessary). This special stack is cleared on exit from each computational routine prior to entering the next opcode.

The two address fields for exception handling introduce further flexibility and transparency in the system. In some cases these fields point back to the general decoder of this subcase, other times to the code that indicates RANK ERROR, etc.. In addition there exists a handler called FORCE which attempts to convert data representations downward, i.e., from real to integer, etc.. Any routine which must have a particular internal type (such as integer subscripts for indexing) uses FORCE as its type exception handler. FORCE will either succeed in doing the appropriate conversion, and (as with FETCH) simply return transparently to the subcase that called it, or else result in a DOMAIN ERROR. (There is another form of FORCE which also attempts to convert downward but returns with an error flag rather than causing DOMAIN ERROR. This version is used, for example, in fetching an axis indicator for dyadic comma, which may legally be either integer or non-integer. The special cases that handle catenation (not lamination) expect to get an integer axis, which might happen to be in real representation internally, as in the case $A,[0.5+0.5]B$. On the other hand, they cannot cause a DOMAIN ERROR should the result not be integer or integerizable.)

ALLOC has many possible actions, depending on the nature of the object in question. For temporaries, space must be found on the appropriate stack (or in the descriptor entry). Named variables require that previous freespace storage (if any) be returned to the system and new space allocated. (Our system in fact attempts to reuse the same storage rather than returning and reallocating.) Furthermore, in some cases data values in free space may be shared between several different data descriptors, in which case a reference count mechanism is invoked in ALLOC. Finally, based on the outcome of the above actions, the various modifier bits may or may not require alteration.

Similar (although less complex) choices exist for STORE and FREE. Rather than describing them in depth, we mention two responsibilities of STORE. Firstly, shared variables are forwarded to the Shared Variable Processor at STORE-time, once their computation has been completed. Secondly, in the case of (assignable) system variables, STORE will cause an appropriate validation routine to be entered. Each

validation routine checks that the system variable meets the restrictions of the language (for implicit use), and if so, normalizes the internal representation (i.e., □IO will be stored as integer, □CT as a floating point "fuzz", etc.) and sets a validity bit on. Otherwise the bit is set off, and the value is untouched, but is still accessable explicitly. All execution routines which implicitly use system variables simply check the validity bit; if it is on, then they can access the value without going through the entire FETCH mechanism each time. (To insure that system variables are accessable directly via the desriptor table, any localization of them is always treated as a "push" by the code generator.)

INTERNAL OBJECT TYPES

All addressable objects (including temporaries) are given an entry in a descriptor table. Each entry occupies a total of 56 bytes, for reasons we soon outline. Objects fall into several categories that require differing treatment. The kinds of objects are:

1) non-shared identifier
2) shared identifier
3) system variable
4) left temporary
5) right temporary.

Several modifiers may apply in addition:

1) long or short data
2) unitialized variable
3) localizable system variable (or not)
4) validity bit for localizable system variable
5) internal system variable name
6) internal shared variable name.

It is clear that if we were to compactly encode the several possible attributes and modifiers of each descriptor table entry into a few bits, we would have to go through a fairly complex software decode procedure for every FETCH, ALLOC, STORE, and FREE. The scheme adopted in order to avoid this expense resembles the code adaptation described above.

Figure 2 shows the layout of a descriptor table entry. We allocate a full word to each of the four fields FETCH, ALLOC, STORE, and FREE. These fields hold the address of the appropriate routine which carries out the required action. Thus the execution routines simply load this address into a

register and BALR directly to the correct handler with no decode overhead. (In fact, we have a kind of 2*24 bit decoder "free-of-charge" in the 370 hardware!)

For example, the code generator initializes the FETCH field for an identifier so that it points to a routine which produces the VALUE ERROR message. When a variable receives a value, this field is changed to a true FETCH routine address. Similarly for a system variable the STORE field points directly to the appropriate validation routine for this system variable. The FREE routine for temporaries points to the routine that lowers either the left or right hand stack as the case may be, and otherwise points to a BR 14 instruction.

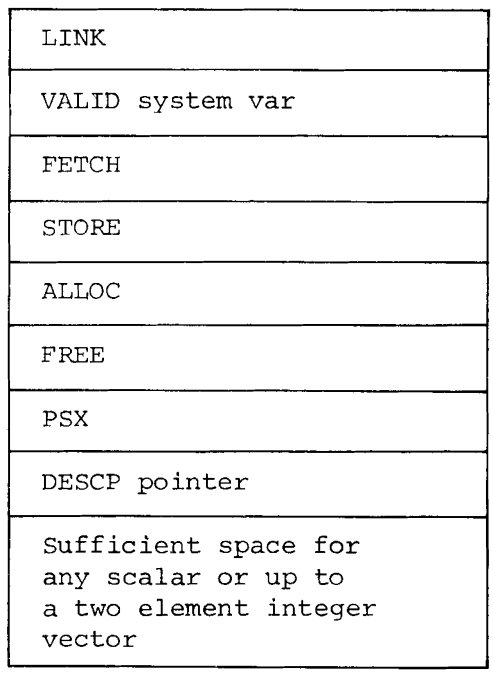


Figure 2. Descriptor table entry

PERFORMANCE

Appendix A presents the instruction trace of the entire interpretation for addition of two integer scalars. In the example we follow the shortest path through the code, i.e., we assume no type conversions are necessary, no overflows occur, etc.. The instructions shown are complete; i.e., if we were to execute several routines in sequence, the BNEP at the end of one branches to the LM of the next.

Measuring (by timing or estimating) the scalar addition routine shows that it is 5 to 10 times faster than the comparable routines in VS APL or APLSV. There are two points we can make here. To begin with, it is hard to envison a further reduction by a factor of 2 in the number of instructions without using direct compilation (or microcode support, as will be discussed later). Also it is clear why the addition of further overhead (such as space management for small objects) can seriously impact the current performance.

Before proceeding to discuss a comparison based on a practical workspace, we would like to present the flavor of the performance differences we measured for "large" aggregates. In principle, all these ratios should be near one, and they demonstrate how far one can go by optimizing the inner loops. Table 1 shows the ratio of times measured on a 370/168 between APLSV (Version 1.2) and our host when the number of elements was non-trivial. (The exact number of elements was selected based on observing the behavior of the ratio as we increased the size of the objects.)

Table 1. Comparison of 370 host to APLSV (V1.2) for large aggregates

Expression	Approximate Performance ratio
⍳Is	1
Ia+Ia	10
Ra+Ra	5
Iv⍴Ba	500
Iv⍴Ca	5
Iv⍴Ia	5
Bv∈Bv	200
Rv∈Rv	3
Bv≤Bv	1000
Bv∧Bv	10

Key: Is-integer scalar, Ia-integer array, Ra-real array, Iv-integer vector, Ba-boolean array, Ca-character array, Bv-boolean vector, Rv-real vector.

Because of the limited nature of our implementation of the 370 host, it was not possible to evaluate our system on a large sample of APL workspaces or testcases. Fortunately, there was a particular workspace which was compilable as is, and represented an example of an application that is convenient but expensive to run in APL. The program was written by one of the authors (E.J.S.) and is a simulation of a 32 multiprocessor system designed by Flynn, et al [13]. One function (SIMULATE) actually describes the machine; the remaining functions constitute a general package for time-driven simulations. The package was intended as an example of how quickly and easily one can construct a simulator in APL, and was used in a course on Computer Architecture. The flexibility of the package was readily shown, but the Flynn machine simulation is quite costly to run. The sample used takes about 70.7 seconds of 370/168 CPU time using APLSV Version 1.2.

The compiled version of SIMULATE was run on 370/168 and took 11.2 seconds of CPU time. The ratio of APLSV (V1.2) time to the compiled version is 6.3; this is a reduction of 84% of the original CPU time.

A GPSS-V version roughly equivalent to the function SIMULATE was done by an experienced GPSS programmer. The GPSS solution took 3.6 seconds while the compiled APL version took 11.2 seconds. The APL version was immensely easier to develop, modify and debug, and we believe that these two times are reasonably close, considering the differences in algorithm, programmers, etc.. Certainly the ratio of 70.7 to 3.6 seconds (i.e. about 20 to 1) is a most unfavorable statistic for APL advocates; a factor of 3 is much more attractive.

FUTURE DEVELOPMENT

As has been mentioned, our host implementation is incomplete, to the point that it is extremely unlikely that any given workspace is currently executable. Some of the omissions present no further problems other than simply coding, e.g., the monadic scalar functions, grade up and grade down, format, etc.. The most significant category of omissions are the APL operators: reduction, scan, inner and outer products. The primary issue here is which cases (of which primitives) are worth including in fully optimized form. Producing the code for a particular one is straightforward once the decision has been made. The remaining question is how to treat all the unoptimized

cases, for instance, of inner products, where many hundreds of cases exist, but are rarely, if ever, used. Here we expect to revert to a generalized execution scheme, as is done in traditional APL interpreters.

Another possible extension to our work is substantial further optimization by the compiler phase. There are technical problems due to uncertainties about control flow in the static APL programs but, assuming we can overcome them, the possibilities for a far better compiler are exciting. We also assume that the compiler is permitted to reorder code where it is safe, i.e., only error situations may have different behavior from the formal specification of APL. We foresee incorporating constant propagation, common subexpression elimination and dead variable analysis as some of the reasonable optimizations to use. (In fact, some of the Abrams-like redundant computation elimination [12] could be performed at the source level.)

We strongly recommend procedure integration (i.e. in-line substitution of the procedure body) as advisable. For example incorporating the function IF of workspace SIMULATE in-line yields a substantial gain in performance. This was done manually, and the compiled version took 9.0 seconds to execute, a saving of 20% over the version without procedure integration. This faster version is 7.9 times faster than APLSV (V1.2) executing the original workspace.

Our last suggestion is that the compiler recognize and translate larger expressions (which the common subexpression eliminator looks for in any case), such as ⍴⍴, ⍳⍴, 1↑⍴, $I \leftarrow I+1$, etc., as primitives of the intermediate target machine. We feel that a small number of special cases would cover the popular situations and provide still further optimization. This technique could also capture (at compile time) expressions where the loop merging techniques of Abrams could be used without run-time overhead.

The last avenue of further development we suggest concerns microcoded (or hardwired) support for our system. In an attempt to understand performance and find any candidates for special optimization, we ran the host interpreter under a (software) sampling monitor which produces a profile of the interpreter execution. No subcase took more than 1% of the total execution time. The only peaking that was significant was in the common handlers, FETCH, ALLOC, etc.. In total these utilized somewhat over 20% of the CPU time, and this does not include the overhead for calling and setting up the relevant registers before entry. Perhaps

there is further optimization possible on the 370 code, but it is surely small. The amount of effort required to implement these few routines in microcode is quite small, and they are easily cast as 370 machine level instructions.

For example, in the trace of Appendix A we see that the first 32 instructions (of 45 in all) can be replaced by 6 instructions (FETCH, LOAD, FETCH, LOAD, LOAD ADDRESS, ALLOC) which would operate at essentially machine cycle time plus several memory fetches. This form of special support is certainly more feasible across the line than the inclusion of a very large package such as the APL Assist.

SUMMARY

We have presented a description of a translator and high performance software interpreter for the APL language. The work described was a feasibility study for an APL compiler. It focussed largely on the questions of expected performance within the framework of as complete an APL language implementation as possible and with no special machine support. The resulting design seems directly adaptable to future new versions of the APL language. There has been no study made of the system aspects of integrating this system with existing interpretive systems, nor on the efficiency of the compiler phase itself.

The system is capable of supporting almost the full language, with the exception of those parts which themselves dynamically modify or construct programs. The system does not require the user to add declarations or otherwise rewrite his workspace.

The system has been partially implemented in order to estimate its performance. The implementation did not utilize any microsupport or postulate any special instructions beyond the standard 370 set; thus it can be utilized on the fastest of the present machine line. The relative improvement compared to existing interpreters varies from one test case to another, and among different existing implementations. The major observed savings were: 5 to 10 times lower overhead for simple operations and varying performance improvements (from 1 to 1000 times faster) when dealing with large data aggregates.

The compiler has been written in APL, and it produces a simple form of machine code for a virtual APL machine. By relaxing somewhat the legal system behavior for erroneous

situations it appears possible to add a wealth of optimizations analagous to those used in compilers such as the PL/I optimizer, etc.. The compiler performs some optimizations based on information provided by the user at compile time. The primary information sought from the user are the names of variables which are shared with other processors, such as TSIO.

The run-time system is a high performance software interpreter which achieves substantial performance gains using adaptive techniques, both at the level of the intermediate code, and in the manipulation of the descriptor table entries. These techniques, as well as other features, are applicable in part to existing APL interpreters.

The present version of the run-time interpreter needs substantial further coding for its completion. There appear to be no major new technical problems in doing so. We have described several techniques by which still further increased performance could be achieved. We believe that the addition of a few new instructions which perform basic operations for the run-time interpreter will substantially improve performance. The modifications we envisage seem implementable even on the highest performance processors since the function of these instructions is not more complex than other existing 370 instructions.

REFERENCES

1. Saal, H. J. and Weiss, Z. An APL compiler. IBM Israel Scientific Center, 1976. (IBM Confidential).

2. Hassitt, A. and Lyon, L. E. The APL Assist (RPQ-S00256). IBM Palo Alto Scientific Center Report ZZ20-6428. Feb. 1975 (IBM internal use only).

3. Compton, M. T. APL in PL/I. IBM Research Report RC4481, IBM Corporation, Yorktown Heights , N. Y. Aug. 1973.

4. Moruzzi, V. L. APL/FORTRAN translations. IBM Research Report RC3644, IBM Corporation, Yorktown Heights, Dec. 1971.

5. Ho, R. L. Routines for translating APL into PL/I and PL/S II. IBM Poughkeepsie Laboratory TR 00.2442, May 1973 (IBM Confidential).

6. Jenkins, M. A. Translating APL, An empirical study. Proc. of APL 75, Assoc. of Comp. Mach., N. Y. 1975, 192-200.

7. Alfonseca, M. An APL-written APL-subset to System/7-MSP translator. APL Congress 1973, North Holland, Amsterdam, 1973, 17-23.

8. Mc Nabb, D. private communication, IBM Los Angeles Scientific Center, (IBM Confidential).

9. Bingham, H. W. Content analysis of APL defined functions. Proc. of APL 75, Assoc. of Comp. Mach., N. Y. 1975, 60-66.

10. Saal, H. J. and Weiss, Z. Some properties of APL programs. Proc. of APL 75, Assoc. of Comp. Mach., N. Y. 1975, 292-297.

11. Saal, H. J. and Weiss, Z. An empirical study of APL programs. Int'l J. of Computer Languages, Vol. 2, No. 3, 47-59, Pergamon Press, Great Britain, 1977.

12. Abrams, P. An APL machine. Stanford Linear Accelerator Center Report 114, Feb. 1970.

13. Flynn, M. J., Podvin, A. and Shimizu, K. A multiple instruction stream with shared resources. In Parallel Systems, Technologies and Applications, editor L. C. Hobbs, 251-258, Spartan Books, Washington, 1970.

APPENDIX A:

Instruction trace for PLUSISS
(scalar←scalar+scalar; integer)