Program analysis and code generation in an APL/370 compiler

by Wai-Mee Ching

We have implemented an APL/370 compiler which accepts a subset of APL that includes most language features and a majority of APL primitive functions. It produces System/370 assembly code directly to be run independently of an interpreter. The compiler does not require variable declarations. Its front end, which is machine-independent, employs extensive type-shape analysis based on a type-shape calculus of the primitive functions and global dataflow analysis. Its back end does storage mapping, code generation for various primitive functions, and register management. For one-line functions, compiled code executes at 2–10 times the speed of the interpreter. On programs with many iterations, the execution time of the compiled code not only is dramatically better than that of interpretation, but is actually fairly close to that of FORTRAN. This removes the performance penalty of APL in computation-intensive applications.

©**Copyright** 1986 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

1. Introduction

In the spring of 1983, the author proposed a project to study the automatic extraction by a compiler of the inherent parallelism in high-level language programs. We could have started from new dataflow languages, such as Id and VAL (see [1]), which are specifically designed for parallel execution. However, the disadvantage for such a study of any new language not in wide use is the absence of a ready pool of real-life programs for experimentation. On the other hand, the use of a traditional sequential language such as FORTRAN requires a highly sophisticated special system such as Parafrase (due to Kuck at the University of Illinois) to reconstruct the parallelism out of sequentialized programs. Also, some of the inherent parallelism may be lost beyond retrieval during the serial coding process.

Among existing widely used high-level languages, APL is singularly well suited for our purpose because coding in APL does not involve unnecessary serialization. Moreover, use of arrays and high-level primitives generally causes a well-written APL program to have large basic blocks, which is very beneficial for parallel scheduling. (A *basic block* is a segment of code such that, if the first instruction is executed, all instructions in the segment must also be executed in linear order.) Although APL has been derided as being difficult to read, or worse, encouraging "unstructured programming," it nonetheless has a large and enthusiastic following. It has been used for many years in a wide range of fields, such as finance, engineering, and design automation, but rarely in computation-intensive scientific applications. Its interpreter provided a marvelous programming environment long before research on programming environments became fashionable. The only problem is that no APL compiler is available, thus depriving us of any low-level code to study for rearrangement and scheduling. So, we decided to implement an APL compiler from scratch.

By the middle of 1984, the front end of the compiler was basically finished, and by the end of that summer a basic back end that could generate E-code (machine code for a hypothetical multifunction parallel machine; see [2]) was implemented. Since each E-code instruction had already been implemented as a macro on System/370, we were running compiled APL on a 370. However, due to the architectural mismatch between the E-code and the 370 uniprocessor, the observed efficiency of the code on a small number of examples was not as high as expected. Therefore, towards the end of 1984 we decided to implement an APL/370 back end designed to allow a smooth transition to the production of code for machines having 370-compatible vector architectures. The compiler now produces 370 code which can run under IBM's two operating systems, VM and MVS. We are also preparing to modify the low-level code-generation functions in the back end to permit utilization of the new IBM 3090 Vector Facility. In addition to generating vector code, we also plan to implement multitasking in the near future, and to do experimentation on code generation for some experimental parallel machines.

APL is quite difficult to compile because its very late binding and high-level semantics cause difficulties for traditional compiler technology which seem to preclude an efficient compiler. There are many papers on this topic, but usually no implementation on real machines was done. Recently, two implementations of APL compilers were reported: that due to Budd [3-5], where a modified (concerning declarations and scope rules) version of APL is translated into the C language and then compiled; and that by Wiedmann [6] and Weigang [7], where APL is compiled directly into System/370 code. However, in the second case, the generated code exists in an interpreter environment in the sense that it has to mesh with an existing interpreter but can also call the interpreter to execute portions it cannot compile. Hence, we believe that the work to be reported here is the first traditional compiler for APL in the following sense. First, it compiles a substantial subset of APL without needing extra language features, such as variable declaration, and it strictly preserves the original language semantics within the subset (see the Appendix in [4] for a comparison). Second, the code produced by this compiler executes independently of the interpreter. Third, it compiles directly to machine language, unlike APL compilers that convert from APL to some other high-level language. Hence, our approach certainly entails more effort than translating APL into C or FORTRAN. On the other hand, once such an APL compiler becomes available, it can be an excellent tool for studying the problem of automatic extraction of inherent parallelism by a compiler from high-level language programs. Since APL uses natural vector notation and provides a very large group of high-level compound functions as primitive operations in the language, it will be easier to map APL programs to vector and parallel machines than those written in a traditional sequential language such as FORTRAN. This indeed was the original motivation for this project. An alternative approach, in which APL is translated into FORTRAN, is represented by the work of Driscoll and Orth [8], which also appears in this issue.

In this paper, we first summarize the restrictions of our APL compiler; we expect most scientific and engineering APL users can live with them comfortably. Next we briefly describe how to use the compiler and run compiled code. We then discuss the program analysis component of the front end, where the type-shape calculation and stream-grouping are unique to APL compilers, but where the global dataflow analysis framework, on which the final type-shape analysis is based, is common to other optimizing high-level language compilers. In the next section, we describe the basic code-generation model of the compiler back end (which is machine-dependent), the storage mapping of variables, and the register management scheme. Finally, we give data on the execution time of some compiled code in comparison with the execution time of the interpreter as well as with code generated by VS FORTRAN on corresponding FORTRAN programs.

2. Language restrictions and the compiling procedure

The difficulties of compiling APL are well known. Our basic philosophy is to impose a minimal number of restrictions to make the language subset compilable, but maintain the distinctive features of APL which most users find attractive (e.g., no declarations of variables are required). Our compiler imposes three kinds of restrictions on APL: fundamental restrictions, design restrictions, and implementation limitations. The restrictions of the first kind are fundamental to our approach to APL compilation in the sense that without them the present scheme would not work. For example, we could not have stable parse trees if the execute function were not excluded, and dynamic creation and deletion of user-defined functions would make static analysis impossible. Any APL compiler not relying on help from an interpreter cannot function without these restrictions. From our contacts with scientific and engineering users of APL, we discovered that most of their applications already conform to these restrictions and can be compiled without change. The fundamental restrictions are the following:

1. The compilation unit must be self-contained; i.e., code in the unit cannot refer to functions or variables, except

inputs, not defined in the unit. (We call a group of functions intended to be compiled together a *compilation unit*, which can be a whole workspace.)
2. A function name cannot be used as a variable name or a label name anywhere in the workspace.
3. ⎕*FX* which creates a function from a character matrix (variable) cannot be used, and ⎕*EX* which erases an item can only apply to variables.
4. ⍎ can only apply to B/'S' (which means if B is true execute S), with B a scalar boolean variable or a scalar boolean parenthesized expression and S a directly executable statement.
5. ⎕ input is restricted to constant expressions.

The design restrictions are those features we did not take into consideration during our design. Unlike the fundamental restrictions, there is no apparent reason that they cannot be removed with an expanded effort based on the present approach. These restrictions represent a compromise between the need to limit our implementation effort and the desire not to inconvenience APL programmers who wish to use the compiler. There are three groups of design restrictions:

- First, we do not consider shared variables, and most system features of APL, such as ⎕*CR*, ⎕*NC*, ⎕*PW*, ⎕*AI* (dealing with the canonical representation of a function, the name classification of character data, print width, and account information, respectively), are excluded. The atomic vector ⎕*AV*, representing the character set of the system, is included, but ⎕*IO*, the index origin used to indicate whether we count beginning from 0 or from 1, can be chosen only once for the whole compilation unit.
- Second, for the sake of a simpler implementation, we restrict the target of a branch to be one of the following forms:

  a. 0
  b. 0,numeric-expression
  c. 0×numeric-expression
  d. label
  e. label,numeric-expression
  f. boolean-expression/label-list
  g. (label-list)/[index-expression]

- Third, a variable of general type (i.e., a variable used both as numeric and character in a basic block) is not treated.

The implementation limitations are those language features which are designed to be handled by the compiler but have not yet been implemented due to our manpower limitation. These restrictions will gradually be removed as we make progress. They include the following:

1. The rank of a variable has to be a compile-time constant and may not exceed 7.
2. The primitive functions ⍕\,[*x*]⌽⍉⍋⍒?⍟!⌹, i.e., the format, expand, laminate, rotate, transpose, grade up and grade down, general logarithm, factorial and binomial, and matrix division functions are not supported at this moment.
3. Certain combinations of derived functions considered to be used very infrequently are missing right now.
4. Character input-output is not supported at present.

The compiler is divided into a front end and a back end, both of which are written in APL. Hence compilation is carried out in the APL interpreter environment, and the procedure is quite simple. To compile a unit of functions in an APL workspace, the user first loads the front end of the compiler. He then copies the group of functions from its workspace and issues the command

```
'mainfn typs' COMPILE qdio shape1 shape2
```

where *mainfn* is the name of the main function in the compilation unit (all other functions in the compilation unit are called directly or indirectly by that function); *typs* is a null, one-, or two-character string indicating the storage types of the parameters, if there are any, of the main function: B = boolean, I = integer, E = floating-point, and C = character; *qdio* is either 0 or 1, representing the intended ⎕*IO* for the compilation unit, and *shape*1 and *shape*2 are numerical vectors indicating the shapes of the main function parameters. A scalar is represented by a 0, a vector is a a 1 v1, where v1 is the length which is a ¯1 if it is not known at compile time. Similarly, a matrix is a 2 d1 d2, with d1 and d2 the dimensions. The user then copies in the compiler back-end workspace and issues the command

```
CODEGEN 'mainfn'
```

This generates a CMS file named "mainfn" ready to be assembled into a load module. One or two input files are prepared if the main function has parameters (each input file must be headed by records indicating the storage type, the rank, and the shape before the values of the argument), and then the compiled code is executed. The generated 370 code can also be shipped to the MVS system to be run under TSO provided the input files are in the MVS environment.

We note that, although ⎕*SVO* is not supported at present, the most common use of this system function is to read a file into the APL environment. Since execution of the compiled code starts with reading into memory one or two input files if the main function for compilation has one or two parameters, the current setup can be easily modified to read in additional input files to cover the need for using ⎕*SVO* for input purposes. The newer features of APL2 [9] are not addressed by our compiler. However, the APL2 interpreter has the capability of calling assembly language modules. Therefore, a programmer can encapsulate the computation-intensive part of his program in a self-contained unit not using the newer features and have it compiled to be called by other parts of his APL2 program, thus benefiting both from APL2 and the compiler.

3. Compiler organization and program analysis

The front end of the compiler has three components:

1. Lexical scanner and parser.
2. Global data flow analyzer and constant finder.
3. Type-shape analyzer.

The lexical scanner and parser produces flow graphs (one for each defined function), a call graph, a name table, and parse trees (one for each line). A name can be a function name, a variable name, or a label. This component also sets up the function table, variable table, constant tables (one for numeric and one for character constant), shape table, and branch table. The lexical scanner is actually integrated into the parser as in the case of a typical Pascal compiler. It uses a novel parsing algorithm which goes from left to right; parse tree nodes are built up in the reverse order from the one in which they would be executed. This algorithm combines the two passes, one to scan the source code from left to right and one to parse the tokenized line from right to left, needed by a typical APL parser. Primarily due to the elimination of an additional pass, our parser was found to be about eight times faster than the parser, also written in APL, implemented by J. J. Girardot in 1981 using the traditional two-pass method. The algorithm is nonrecursive and uses a two-symbol look-ahead technique and shifts between two parsing states.

The second component of the front end, the global dataflow analyzer, is the most elaborate from an algorithmic point of view. It is built on the important work of Allen and Cocke [10] on the interval-based method of global data flow analysis for program optimization.

The third component of the front end, the type-shape analyzer, is unique to an APL compiler. Other high-level language compilers, with the exception of SETL, do not have, and do not need, this component since variables are declared. The basic algorithm was devised by the author in 1981 [11]. Since variables are not declared, the main task for the front end, other than those common to most other high-level language compilers, is to analyze the number of variables used and to infer their storage types and shapes from their usage. Tennenbaum [12] is the first to have reported type determination work in compilers; it was implemented in the SETL language system (the work in [13] is of a theoretical nature, as a representation of the computation states at each node is quite impractical for real languages on machines of present memory capacity). We note that in Budd's modified APL compiler [3] each variable is declared with its storage type and carries a general shape box which can be manipulated by the generated code. This removes the necessity for compile-time type-shape analysis; nevertheless, compile-time analysis still helps. In Wiedmann [6], most work seems to be in finding the ranks of the variables. Our compiler attempts to find the types and shapes (not necessarily the exact shape, but as much information about the shape as possible) of all variables, so that they can be allocated either on the stack or in the heap at compile time without requiring excessive indirect access at run time. The worst case is that the front end of the compiler finds some variable to be of general type or of general shape; that is, nothing specific can be said about a particular type or shape. The front-end compilation process will continue to its end. But when the back-end compilation process is started, it will ask the user to supply the type (shape or rank) of any variable assigned a general type (a general shape) because of our design (implementation) limitation. As it turns out, most variables (with the exception of the parameters of the outermost function) in real-life programs that have been processed by our front end so far have been scalars, vectors, or matrices whose types and shapes can be inferred from the program text most of the time. We have not yet encountered a case of a variable assigned a general type in programs processed by the front end. We have encountered some cases of general shape resulting from loss of shape in our analysis in an interprocedural setting. As can be seen from setting of the storage formats of variables in the next section, we do intend to handle variables having the general shape, since the admission of the laminate function implies the admission of variables of changing ranks. But our current implementation effort has not yet covered this. That means, for the moment, that the back end cannot generate code for a program having a variable whose rank cannot be determined at compile time.

There are five storage types of variables, each represented by a number: boolean—0, integer—1, real—2, character—3, and general—4 (a variable of type 4 is not treated by the back end at present). Since the number of possible shapes is infinite, a shape is represented by a (variant) record of three fields: the rank, the first dimension, and a pointer to where the rest of shape (i.e., 1 ↓shape) is stored. Each field can take a symbolic value which may be the unique value "unknown." When the rank field takes the value "unknown," the third field indicating the shape tail must also be "unknown." There is a unique shape, denoted by the number 6, representing the most general shape. We associate a type and a shape with each node in a parse tree, each variable in the variable table, and each constant in the constant tables (one numeric and one character). For any defined function (i.e., a procedure), the initial source of information on types and shapes is from constants and parameters, except for the main function, where they are supplied as arguments to the compilation procedure. We know the types and shapes of constants but not those of parameters other than those of the

Table 1 Properties of primitive APL functions.

fn	*right-typ*	*left-typ*	*resl-typ*	*right-shape*	*left-shape*	*resl-shape*
+	2	2	6	*A*	*B*	*R*
=	4	4	0	*A*	*B*	*R*
ι	1		1	*S*		*V*
ϕ	4	1	5	*A*	*W*	*R*

main function. Hence, we give parameters the general type but assign each a unique shape number to denote an unknown shape (a method inspired by the value-number method in [14, Ch. 6]. Later each shape number is either replaced by a more concrete shape or remains unknown. In the latter case, we chain all nodes with this number to one general shape.

Both constant propagation and local type-shape analysis use a modified Schorr-Waite algorithm (see Algorithm E [15, Section 2.3.5] for the basic algorithm) to traverse a tree from the rightmost node to the root nonrecursively. For constant propagation, in the first pass through trees we collect, as the first group of constant variables, all those variables assigned a literal constant only once. In the second pass, all single-assignment variables are checked to see whether the expressions assigned to them involve only literal constants and constant variables from the first group. This method was found to be quite effective for APL programs.

The local type-shape calculus is based on the fact that each primitive function has certain type and shape requirements and produces a result whose type and shape are a function of the type(s) and shape(s) of its argument(s). This semantic information about the transformations on types and shapes effected by all primitive functions in APL is encoded in a table such as that shown in **Table 1.**

The table shows the properties of primitive functions; for example, the scalar functions = and + accept arguments of compatible shapes and produce a result of the same shape; the function = accepts arguments of any type (4) and produces a boolean result; while the function + accepts numerical arguments and produces a numerical result which is the "maximum" of the two (6); i.e., a real argument and an integer argument give a result of a real type. However, the addition of two arguments of boolean type results in an integer type because there we take the minimally required type into consideration. The type-shape behaviors of mixed functions are much less uniform. For example, the monadic ι (ι*N* generates a vector of integers from 1 to *N*) accepts only integer scalars, whereas dyadic ϕ(rotate) accepts arrays of any type and shape for its right argument, and the resulting type and shape are the same as those of the right argument, indicated by a 5 and *R* (see [11] for more details). The flexibility provided by the language in allowing one scalar argument to be reshaped to that of the other in many cases causes great difficulty in the analysis because it represents a nonuniformity of the rules. The result of type-shape calculation at each node is propagated forward through the tree and through all basic blocks (in a defined function) in a depth-first search order. The constraints that a function node imposes on its argument(s), if not immediately satisfied, are also propagated backward, but only to the extent of one parse tree.

Global dataflow analysis is employed here primarily for the global type-shape determination and the solution to the live-analysis problem. The analysis is based on the reach and live-analysis algorithms in Allen and Cocke [10]. We modified their basic algorithm by using Tarjan's fast interval-finding algorithm [16] to find intervals without flow graph reduction, and incorporated an iterative component to deal with improper intervals—those containing irreducible subgraphs (see [10]). We calculate the solution to the reach problem; i.e., for each basic block we find the set of all definitions that have arrived at the top of that block. This, in turn, enables us to do U-D chaining; i.e., each exposed-use of a variable in a basic block is chained to the list of definitions of the same variable supplying its value from other blocks. Finally, we solve the live variable problem. More specifically, we mark the spot where a value of a variable becomes dead, i.e., either is no longer needed or is soon to be redefined. Global dataflow analysis is done in an interprocedural framework. In particular, definitions in a function can reach its calling function and vice versa.

In the global type-shape analysis we OR the types and shapes of the definitions chained to a particular exposed-use to get a new type and shape of that use before we begin a new round of local propagation. This is done iteratively until the types and shapes of all variables stabilize. They will stabilize because the OR function on types and shapes to be described below is *monotone nondecreasing* on the lattices of types and shapes ordered by their generality (see [11]) and there the general type and the general shape serve as the greatest elements of the two lattices, respectively. (For practical reasons, we set a limit of 12 on the number of iterations at present.) To OR two types, we simply take their maximum if both are numeric. If one is numeric and the other is character, the result of OR is the general type. To OR two shapes is more elaborate. For example, the OR of two vectors, one with a known length and one with an unknown length, results in a vector of unknown length. There is a shape representing the most general shape, e.g., neither its rank nor any of its dimensions are known, which serves as the universal cover of the OR of shapes.

The front end also finds all *streams* in all parse trees. A stream is a group of consecutive scalar primitive function nodes where the output of one feeds the input of another so that the shapes of the nodes are all the same. A simple example is

```
A←B+C←D×E
```

The recognition of streams helps to reduce the need for intermediate storage and saves unnecessary stores and loads.

Table 2 shows the distribution of time (in CPU seconds on an IBM 3081) spent by each of the three components of the front end for three workspaces with their numbers of functions and noncomment lines indicated. The first is a printer simulation program, the second does topological folding for PLAs (programmable logic arrays), and the third does graph statistics. The first two are compiled by the front end without modification, while the third is taken from a package by stripping away the parts involving auxiliary processors (device drivers).

The time could be substantially reduced if we eliminated the extensive printouts which help compiler debugging and program analysis.

4. Code generation and the execution time of compiled code

The back end accepts the parse trees, flow graphs, and various tables, and it generates 370 assembly code. We use the traditional Pascal-like stack and heap management of memory. The back end first decides the storage formats of variables based on their shapes. There are five storage formats for five cases:

0. A scalar or one-element vector.
1. An array with its shape completely known at compile time.
2. A vector of unknown length.
3. A nonvector array of known rank but unknown shape.
4. An array with unknown rank.

The last one corresponds to the most general shape, and is not now supported by the code-generation functions. The back end calculates the stack length for each defined function and assigns displacements for variables. The stack length of a defined function is the sum of the stack lengths of its local variables, which is 4, 8, and (8+4-rank) bytes for variables of storage format 0, 2, and 3, respectively; it is the byte length needed to store the values of a particular type for variables of storage-type 1. It also reserves system areas, such as input-output buffers and four vector buffers (one for each type), and it fills in the constants. The allocation of storage, base registers for functions, and the function call mechanism are designed to handle recursion.

Table 2 CPU time in seconds needed for the three front-end components to process three sample programs.

WS name	*Funcs*	*Lines*	*Parsing*	*Dataflow analysis*	*Type-shape analysis*
`SIMPLE`	9	74	3.48	5.46	10.79
`CUT`	4	245	9	5.53	29.76
`PLOTL`	22	526	34.3	41.82	44.83

The back end then generates code for each defined function in the compilation unit one basic block at a time. Each basic block consists of one or several parse trees. To generate code for each parse tree, we walk through the tree from the lower right corner to the root using a semi-recursive algorithm. During the tree walk, the main code-generation function calls various low-level code-generation routines, one for each primitive function as it encounters the various function nodes.

The computation state of the machine is represented by the set of 20 System/370 machine registers (16 general-purpose registers and four floating-point registers) and four vector buffers. The convention of vector buffers is peculiar to APL and is borrowed from our previous work on an APL compiler generating E-code. This convention introduces a conceptual simplification for our code generation. The register management is quite simple and conventional. We let scalars (and one-element vectors) stay in the registers as long as they are alive. When we run out of registers during code generation, we push some variables out of the registers (either without necessarily carrying out a store or by actually storing the contents of some registers in a backup store). Those in the backup store will be restored after each unit of code corresponding to one APL operation is completed. We also generate code for segments like

```
A←B+C
```

in an integrated fashion. That is, if B and C are two arrays, part of the result of $B+C$ is stored into A as soon as it is generated without staying in any intermediate storage.

The quality of the code generated is surprisingly good. It not only represents a significant improvement over that of the interpreter but is also quite close to that of FORTRAN on some of our test cases. In order to compare the execution time of the compiled code with that of the interpreter, we first measure the times for two (basically) one-line examples. We all know that the interpreter becomes much less efficient on programs with loops. Hence, it is easy to claim for an APL compiler, which is somewhat faster than the interpreter, any desired number as a speedup ratio by applying the compiler to a program with a very large loop count. But with one-liners, there is no comparable advantage for the compiler in simply increasing the size of the input data (say, a vector), because the interpreter becomes more efficient when the arrays it processes become larger. Hence, in a certain sense, comparisons using one-liners measure the intrinsic speedup of execution that a compiler can provide over that of interpretation. When APL compilers become widely available, there certainly will be people accustomed to FORTRAN (or Pascal) who will write sequentialized APL code. However, to claim a dramatic improvement in performance on behalf of the compiler over the interpreter based on that code is a bit misleading because people knowledgeable in APL do not write code for the interpreter in that way. Nevertheless, this kind of speedup is still important for the popularization of APL. Only that, the true measure of the quality of an APL compiler in these cases, should be a comparison with that of code produced by a FORTRAN (or Pascal) compiler because the new users that an APL compiler can attract are from the FORTRAN camp. Thus, we also provide some comparisons with FORTRAN compiled code (on similar problems) in the Appendix. Execution times are all in milliseconds, and input-output times are excluded except in the first example, where there is an explicit display. The interpreter is VS APL and the FORTRAN compiler is VS FORTRAN, all running, including our compiled code, on an IBM System 370 Model 3081.

Finally, we note that such standard code optimization procedures as common subexpression elimination, strength reduction, and loop-invariant code motion are all missing from our compiler. Yet the compiler does reasonably well without them. The reason, we believe, is that these techniques are needed by traditional sequential languages such as FORTRAN and Pascal to reduce the number of machine instructions in tight inner loops, mostly involving arithmetic computations of vectors and arrays and their index calculations. In APL, these are treated in the code routines for arithmetic primitive functions, and not many index calculations appear in the source code. The major concerns of the back end of our compiler are so different from those in compilers for traditional scalar languages like FORTRAN or PL/I that not only was there little duplication of effort in implementing this APL/370 back end with other scalar language compilers, but it also pioneered new methods of code generation and register allocation.

5. Conclusions

We have implemented an APL/370 compiler for a substantial subset of APL, one comprehensive enough for scientific and engineering applications. The compiler applies the basic program analysis framework developed by Allen and Cocke for (sequential language) program optimization to the unique needs of the APL language. The compiled code executes at a speed 2–10 times that of the interpreter on one-line functions. On programs with large amounts of iteration the execution time of the compiled code is not only a dramatic improvement over that of interpretation but also quite competitive with corresponding code produced by a FORTRAN compiler. This makes APL a new member of the family of languages with compilers, and hence a suitable tool for computationally intensive scientific applications. It also introduces a new perspective to the compiler-oriented approach to parallel processing, since automatic extraction of inherent parallelism from APL programs should be easier than from FORTRAN programs.

The compiler was written in APL and has approximately 4000 lines of code for the front end and 8000 for the back end.

Appendix

- *Example 1*

```
   ∇D REDUCTE; A;B;C;F
[1] F←2.3
[2] C←+/B←F×A←D÷E
[3] ⎕←⌊C
∇
```

Length	*Interpreted*	*Compiled* (1 ¯1 1 ¯1)
5	4	2
20	4	2

- *Example 2 (Finding primes up to N, ⎕IO=1)*

```
   ∇Z←PRIME N;V
[1] Z←2, (~V∊(2↓⍳⌊N*0.5)∘.×2↓⍳⌈N÷3)
                          /V←1+2×⍳⌊(N-1)÷2
∇
```

N	*Interpreted*	*Compiled*	*FORTRAN*
200	9.5	1.75	2
500	75.75	5.5	6.25

- *Example 3 (Polynomial product, A and B coefficient vectors, ⎕IO=0)*

```
   ∇C←A POLYPROD B
[1] C←((RA←⍴A)+(RB←⍴B)-1)⍴0
[2] I←¯1
[3] L0:→(RA=I←I+1)/0
[4] J←¯1
[5] L1:→(RB=J←J+1)/L0
[6] C[V]←C[V←I+J]+A[I]×B[J]
[7] →L1
∇
```

Length	*Interpreted*	*Compiled* (1 ¯1 1 ¯1)	*FORTRAN*
5	9	<1	<1
75	1501	8	6.2

- *Example 4 (Heapsort, ⎕IO=1)*

(code-generation time was 4.879 s)

```
   ∇Z←HEAPSORT A
[1] L←(⌊(N←⍴A)÷2)+1
[2] R←N
```

```
[3] WHILE:→(L≤1)/WHILE2
[4] L←L-1
[5] SIFT
[6] →WHILE
[7] WHILE2:→(R≤1)/END
[8] X←A[1]
[9] A[1]←A[R]
[10] A[R]←X
[11] R←R-1
[12] SIFT
[13] →WHILE2
[14] END:Z←A
  ∇
  ∇SIFT;I;J
[1] J←2×I←L
[2] X←A[I]
[3] WHILE1:→(J>R)/LABEL
[4] →(J≥R)/NEXT
[5] →(A[J]≤A[J+1])/NEXT
[6] J←J+1
[7] NEXT:→(X<A[J])/LABEL
[8] A[I]←A[J]
[9] I←J
[10] J←2×I
[11] →WHILE1
[12] LABEL:A[I]←X
  ∇
```

Length	*Interpreted*	*Compiled*	*Ratio*
5	8	<1	
75	235	3	78

• *Example 5 POISSON solver (⎕IO=0, compiled* 2 ¯1 ¯1*)*

```
  ∇Z←SOLVPOIS RMINBU; P;Q;L;M;S;T;V
[1] Z←⍳0
[2] →(2≠⍴⍴RMINBU)/0
[3] P←1+¯1↑⍴RMINBU
[4] Q←1+1↑⍴RMINBU
[5] L←¯4×(1○○(⍳Q-1)÷2×Q)*2
[6] M←¯4×(1○○(⍳P-1)÷2×P)*2
[7] S←1○○(⍳Q-1)∘.×(⍳Q-1)÷Q
[8] POISSON1
[9] POISSON2
  ∇
  ∇POISSON1
[1] S←S÷(+/S[1;]*2)*0.5
[2] T←1○○(⍳P-1)∘.×(⍳P-1)÷P
[3] T←T÷(+/T[1;]*2)*0.5
[4] V←L∘.+M
  ∇
  ∇POISSON2
[1] Z←⌊S+.×((S+.×RMINBU+.×T)÷V)+.×T
  ∇
```

```
  R
0  0  0  0  0   0  0
0  0  0  0  0   0  0
0  0  0  0  0   0  0
0  1  0  0  0  ¯1  0
0  0  0  0  0   0  0
0  0  0  0  0   0  0
0  0  0  0  0   0  0

  BU
10  5  5  5  5  5  10
 5  0  0  0  0  0   5
 5  0  0  0  0  0   5
 5  0  0  0  0  0   5
 5  0  0  0  0  0   5
 5  0  0  0  0  0   5
10  5  5  5  5  5  10
```

We note that 1 ¯1 1 ¯1 means we compiled with the lengths of vector parameters unspecified, and 2 ¯1 ¯1 means a matrix of unknown length. The second example of finding primes demonstrates the APL style of coding; most likely a sequential algorithm with loops would be adopted when using FORTRAN or Pascal. The third example represents the FORTRAN-Pascal style of coding and is very costly for interpretive APL. The fourth example is also of Pascal style, and is unnecessary for APL as it has the grade functions ⍒⍋ to do sorting. But we purposely used that example to test the capability of the compiler for WHILE loops with function calls at different sites. The heapsort example runs for 2 ms using Pascal for a list of 75 items, and we know exactly where we can improve the performance of our code to make it equal to Pascal. The last example is important because the Poisson equation is the kernel of so-called weather code which is extensively studied by various researchers in parallel processing. For this example, with an input of a 7 by 7 matrix, the compiled code executes in 5 ms, compared to 15 ms for interpretation. We note that POISSON does not have a loop.

When we examine the generated code, we also discover that the remaining inefficiency is primarily due to our not treating the scalar case separately in our arithmetic code-generating function and in our branch function. When we do so, we expect to reduce the 8 ms to something very close to that of FORTRAN and to pull even with Pascal on the heapsort example.

Acknowledgments

The author would like to thank Tilak Agerwala and George Almasi for their support of this project, which is quite unorthodox in parallel processing research. He is extremely grateful for the contributions made by Chamond Liu of IBM Poughkeepsie during his sabbatical at Yorktown. Liu implemented input-output routines, the linkage to the

FORTRAN run-time library, and the initial versions of arithmetic primitive functions. The work of Andrew Xu of New York University (currently at MIT) on branches, comparisons, inner and outer products and the work of David Yang of Harvard University (currently at the University of Illinois) are also greatly appreciated.

References

1. T. Agerwala and Arvind, "Data Flow Systems," *IEEE Computer* **15,** 15–25 (February 1982).
2. W.-M. Ching, "A Portable Compiler for Parallel Machines," *Proceedings of the International Conference on Computer Design,* 1984, pp. 592–596.
3. T. A. Budd, "An APL Compiler for the Unix Timesharing System," *APL '83 Conference Proceedings,* 1983, pp. 205–209.
4. T. A. Budd, "An APL Compiler for a Vector Processor," *ACM Trans. Lang. & Syst.* **6,** No. 3, 297–313 (1984).
5. T. A. Budd, "Dataflow Analysis in APL," *APL '85 Conference Proceedings,* 1985, pp. 22–28.
6. C. Wiedmann, "A Performance Comparison Between an APL Interpreter and Compiler," *APL '83 Conference Proceedings,* 1983, pp. 211–217.
7. J. Weigang, "An Introduction to STSC's APL Compiler," *APL '85 Conference Proceedings,* 1985, pp. 231–238.
8. Graham C. Driscoll, Jr., and Donald L. Orth, "Compiling APL: The Yorktown APL Translator," *IBM J. Res. Develop.* **30,** No. 6, 583–593 (1986, this issue).
9. *APL2 Programming Language: Reference Manual Release 2,* Order No. SH20-9227-0, 1985; available through IBM branch offices.
10. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19,** No. 3, 137–147 (1976).
11. W.-M. Ching, "A Design for Data Flow Analysis in an APL Compiler," *Research Report RC-9151,* IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 1981.
12. A. Tennenbaum, "Type Determination for Very High Level Languages," *Courant Computer Science Report No. 3,* New York University, NY, 1974.
13. M. Kaplan and J. Ullman, "A Scheme for the Automatic Inference of Variable Types," *J. ACM* **27,** 128–145 (1980).
14. J. Cocke and J. Schwartz, *Programming Languages and Their Compilers,* Courant Institute of Mathematical Sciences, New York University, NY, 1970.
15. D. E. Knuth, *The Art of Programming, Vol. I: Fundamental Algorithms,* Addison-Wesley Publishing Co., Reading, MA, 1968.
16. R. E. Tarjan, "Testing Flow-Graph Reducibility," *J. Comput. Syst. Sci.* **9,** 355–365 (1974).

Received February 27, 1986; accepted for publication April 30, 1986

Wai-Mee Ching *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Ching received his Ph.D. in mathematics from the University of Toronto, Canada, in 1968. He taught mathematics at Louisiana State University, Fordham University, and Cornell University, and he was a Visiting Member at the Courant Institute of Mathematical Sciences, New York University. From 1977 to 1979, he was a senior member of the technical staff in Perkin-Elmer's Data Systems Division. Since 1979, he has been a Research staff member in the Computer Science Department at the Thomas J. Watson Research Center.