ACKERMAN82

Ackerman, William B. Data Flow Languages, IEEE Computer 15, 2(February 1982) pp. 15-24

Within the field of Computer Science there is a strong interest in parallel processing. As a result, there have been attempts at designing compilers that optimize programs written in conventional programming languages, e.g. the vectorization of compilers for FORTRAN. Moreover, concurrent Pascal, Glypnir, and vector dialects have been designed to facilitate the use of the architectural features of multiprocessor, vector machine, and array processor computer systems. Ackerman notes that "many of these languages or dialects are 'unnatural' in the sense that they closely reflect the behavior of the system for which they are designed, rather than the manner in which programmers normally think about problem solving." That is, these attempts have made the properties of such systems visible to the programmer and hence act as vehicles that the programmer can use to help a compiler uncover the presence of parallelism within a program. It is in light of these points that the author makes the recommendation that applicative definitional languages be considered for data flow systems.

Data flow computers seek to take advantage of parallelism and hence need programming languages that efficiently use such an architecture. The author's recommendation for the use of the applicative definitional languages is based upon a presentation of the characteristics of applicative and definitional languages in respect to their ability to work in a parallel processing environment. The characteristics of data flow languages, based on the above two groups, are expounded upon in detail. These properties are namely: (1) a freedom from side effects, (2) locality of effect (3) equivalence of instruction scheduling constraints with data dependencies, (4) "single assignment" conventions, (5) distinctive iteration notation, and (6) a lack of "history sensitive" procedures. Well chosen examples from the data flow languages Val, Id, and Lucid are used to illustrate the given presentation on each of these properties.

In distinction to multiprocessor systems, data flow systems are designed to execute algorithms with a fine grain of parallelism efficiently. Parallelism is exploited at the level of individual instructions as well as at coarser levels. Thus, the following FORTRAN code would contain the permissible computational sequence of (1,[2 and 3 simultaneously], [4 and 5 simultaneously], and 6)

1 P = X + Y 2 Q = P / Y 3 R = X * P 4 S = R - Q 5 T = R * P 6 RESULT = S / T

Typically, programs contain sections, often far removed from each other, at which computations may proceed simultaneously. However, as the author points out, the exploitation of parallelism at all levels is dependent upon the instruction sequencing constraints which must be deducible from the program itself. That is, the status of each innstruction is kept in a special memory that is capable of executing the instruction when all of the necessary data values have been met. Thus, the programming language for a data flow computer system must meet two criteria:

(1) It must be possible to deduce the data dependencies of the program operations.

(2) The sequencing constraints must always be exactly the same as the data dependencies, so that the instruction execution rule can be based on the availability of data.

As is demonstrated, these criteria can easily be met by utilizing the properties of locality of effect and freedom from side effects.

Locality of effect means that instructions do not have unnecessary far-reaching data dependencies. For the FORTRAN example the temporary variables P, Q, R, S, and T are used. The use of these variables in a fragment appearing elsewhere in the program for some unrelated computation would prevent concurrent execution due to the apparent data dependencies arising from the duplication. In turn, freedom from side effects is necessary to ensure that the data dependencies are the same as the sequencing constraints.

Typical side effects arise in procedures modifying variables in the calling programs and in the way data structures are manipulated in conventional languages. The author's recommended solution is the manipulation of data structures in the same way that scalars are manipulated. The simplest operator to perform the applicative equivalent of modifying an array takes three arguments: an array, an index, and a new data value. The result is a new array containing the new data value in the desired location. Thus, in FORTRAN, A[3] = 0 performed on the array values [3, 1, 4, 1, 6] results in the sequence [3, 1, 0, 1, 6]. Using the above stated method the expression B ;= A[3,0] leaves B with the same final sequence and does not change the array A. This value-oriented approach to arrays calls for the programmer to view an array as values instead of objects residing in static locations of memory. Ackerman notes the outcome of using applicative languages that perform all processing by means of operators applied to values and are thus natural languages for data flow computations:

> Viewing arrays and records as values manipulated by operations allows the parallelism among the operators to be deduced from the data dependencies.

Having demonstrated the suitableness of an applicative programming style and a value-oriented rather than object-oriented execution model, the implications of this style are related to the assignment statement, iterative structures, and error/exception handling. The assignment statement, e.g. S := X + Y, should now be seen as a definition rather than an assignment since the only effect is the binding of a value to the name appearing on the left side of the statement. Such definitional type language statements are suitable for program verification since the assertions made in proving correctness are exactly the same as the definitions appearing in the program itself. This is in distinct contrast to conventional imperative languages where one must follow the flow of control to determine where in the text assertions are true. For iterative structures all redefinitions take place at the boundary between iterative cycles. The author concludes that the desired definitional characteristic and ease of program verification can be found in existing definitional languages. Thus, by using the properties of applicative and definitional languages suitable data flow languages can be created. In the concluding section of this well presented paper the author examines methods of achieving high speed parallel execution on data flow computers or supercomputers.