A variable delay method for improving recognition of parallel processable code in computer programs

R. G. Ward

Department of Computer Science, Stephen F. Austin University, Nacogdoches, Texas 75961, USA

In this paper we present an algorithm that delays the necessary involvement of a variable in the evaluation of an arithmetic expression until the latest possible moment, and at the same time it identifies all possible operations in the expression that may proceed prior to the delayed variable. A method is described for incorporating this algorithm into a scheme for recognising inter-statement parallel processable code in computer programs. The method and the value of the technique are illustrated through analysis of an example program.

(Received February 1973)

1. Introduction

Based on necessary conditions set forth by Bernstein (1966), there have been three major systems written for the automatic detection of inherent parallelism between statements in higher-level languages. These systems include the FORTRAN Parallel Task Recogniser of Ramamoorthy and Gonzalez (1969*a*, *b*), the UCLA FORTRAN analyser of Russell and Baer (1970), and the ALGOL analyser of Bingham, Fisher and Reigel (1967). Baer (1972) has pointed out, however, that these systems have several shortcomings. One is that no attempt is made to discover what Volansky (1970) calls 'hidden parallelism'. Volansky (1970) proposed a method for memory deallocation which would improve the recognition of parallel processable code. His method could prove to be a valuable technique, but as Baer (1972) indicated, there are several severe drawbacks to Volansky's method. The objections mainly involve the inordinate amount of time and storage needed to apply the technique.

The purpose of this paper is to present a technique for discovering another type of hidden parallelism and demonstrate its value. Comparing it to Volansky's (1970) technique, the method that will be described is economical and entirely feasible. However, the benefits accrued by its application may not be as great as that of Volansky's, where for some example programs a saving of up to 50% has been reported (Baer, 1972).

2. Problem and motivation

Given an arithmetic expression E and a set of variables D used within E that are causing a serial ordering of the statement S containing E and those statements T whose outputs include D, it is desirable to have an algorithm which accepts E and D as inputs and identifies all computations in E that may proceed prior to the necessary involvement of the variables in D. The algorithm should produce as output a set of temporary results and a reduced expression E'. The temporary results represent the operations in E which can be initiated before the necessary involvement of the variables in D, and E' is an equivalent expression of E with temporary variables in place of the operations represented by the temporary results.

If E is replaced in S by E', then the computations represented by the temporary results can be initiated in parallel with the operations in T. The net result is a reduction in the total computation time of the operations in the set $T \cup \{S\}$.

Consider, for example, the execution of the following two serially ordered statements in a two processor multiprocessor system.

(*a*) $r = a/b$

⋮

(*b*) $c = 2.*3.14159*r$

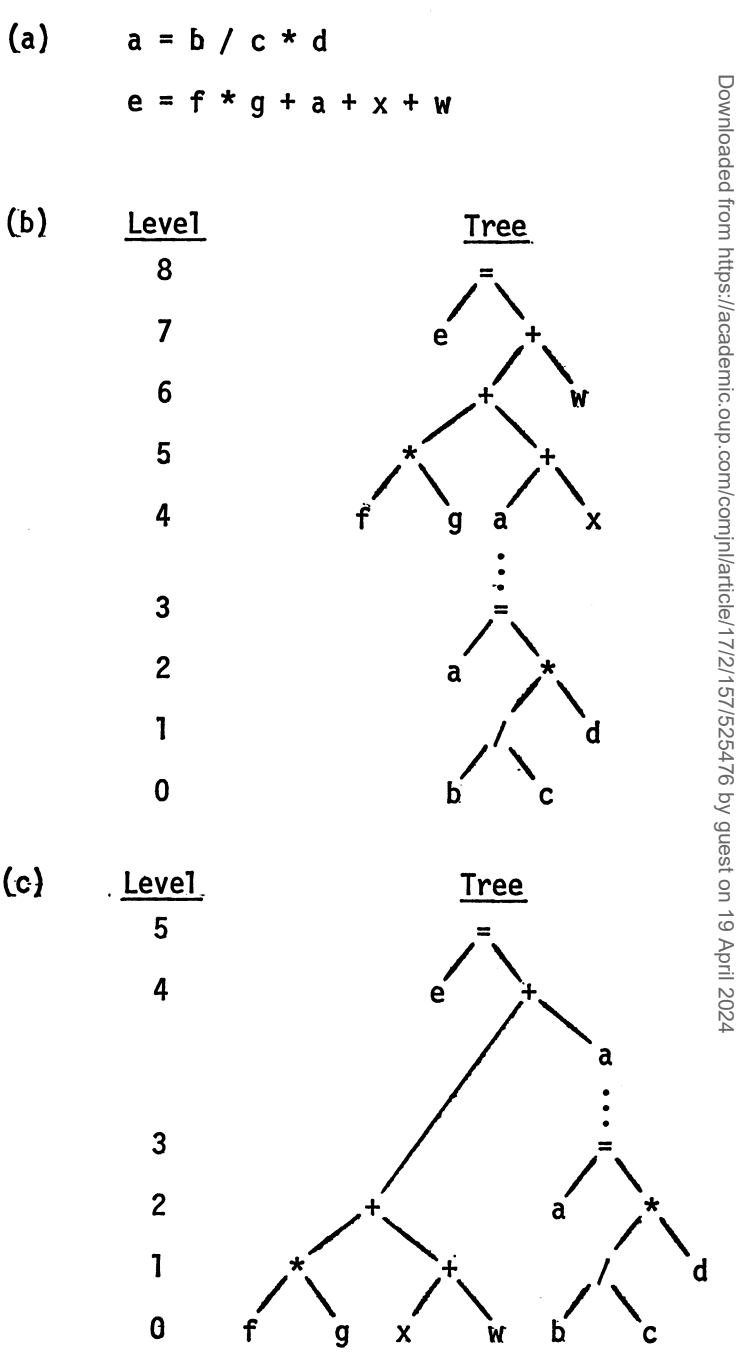


Fig. 1 Two serially ordered statements treed with and without variable delay

Treating =, / and * as operations with each taking one time unit to perform, then statements (*a*) and (*b*) have two and three operations respectively; and since they are serially ordered with each other, they have a total computation time of five time units. This also means, for our purposes, that one of the processors in the system is idle for five time units. If we apply the desired algorithm mentioned above to the expression in (*b*) we have the following results.

(*a*) $r = a/b$

⋮

(*b'*) $t_1 = 2.*3.14159$

(*b*) $c = t_1*r$

Now, statements (*a*) and (*b*) are still serially ordered. However, (*a*) and (*b'*) may be executed in parallel, and the net effect is that statements (*a*), (*b'*) and (*b*) would have a total computation time of four time units on the two processor system. This represents a significant improvement, and one of the two processors would be idle for only two time units.

As another illustration, consider the execution of the serially ordered statements given in **Fig. 1**(*a*) on a parallel processor system like the CDC 6600, the CDC 7600, or the IBM 360 Model 91. In these systems, a program's instruction execution rate is improved by simultaneously performing different instructions on independent functional units at the same time. For this example, we will use Lorin's (1972) pseudo 6600 computer which has two fetch units (FU1 and FU2), one add unit, one multiply unit and one divide unit. The pseudo machine's timings and characteristics are derived from the CDC 6600 (Control Data 6600 Reference Manual, 1969) and basically are as follows. Instructions are spontaneously made available from an instruction stack with no look-ahead feature, and each has a one-cycle decode time. There is a one-cycle stabilisation period (*S*) after an instruction is decoded and before it is released to a functional unit. A memory reference requires five cycles, and we will assume there are no memory contention delays. After a functional unit develops a result, there is a one-cycle interlock period (*I*) before the functional unit becomes available. After an instruction is decoded, it is not released for execution if the appropriate functional unit is not available. With no instruction look-ahead feature, this will delay the decoding of the next instruction until the needed functional unit becomes free. Also, an instruction designating a register as a result register is not released for execution if that register is currently designated as a result register of some other instruction being executed. Finally, it is assumed that a source register (which is not a result register too) becomes available one cycle after the activation of the instruction using it.

In Fig. 1(*b*), a tree is given for the operations specified in the statements of Fig. 1(*a*). Since these statements are serially ordered, the first one must be completed before the second one is initiated. Therefore, the operations for the first statement appear in levels 0–3 of the tree, and the operations for the second one appear in levels 4–8. In **Fig. 2**, a chart is given representing the execution of machine code that corresponds to the tree in Fig. 1(*b*). Time units are laid out horizontally in the chart, and a line segment in a time unit (for a given functional unit) indicates that the functional unit is active during that period. Parallel activity is indicated by multiple line segments within a given time unit. From Fig. 2 we see that a total of 101 machine cycles are required to execute the two serially

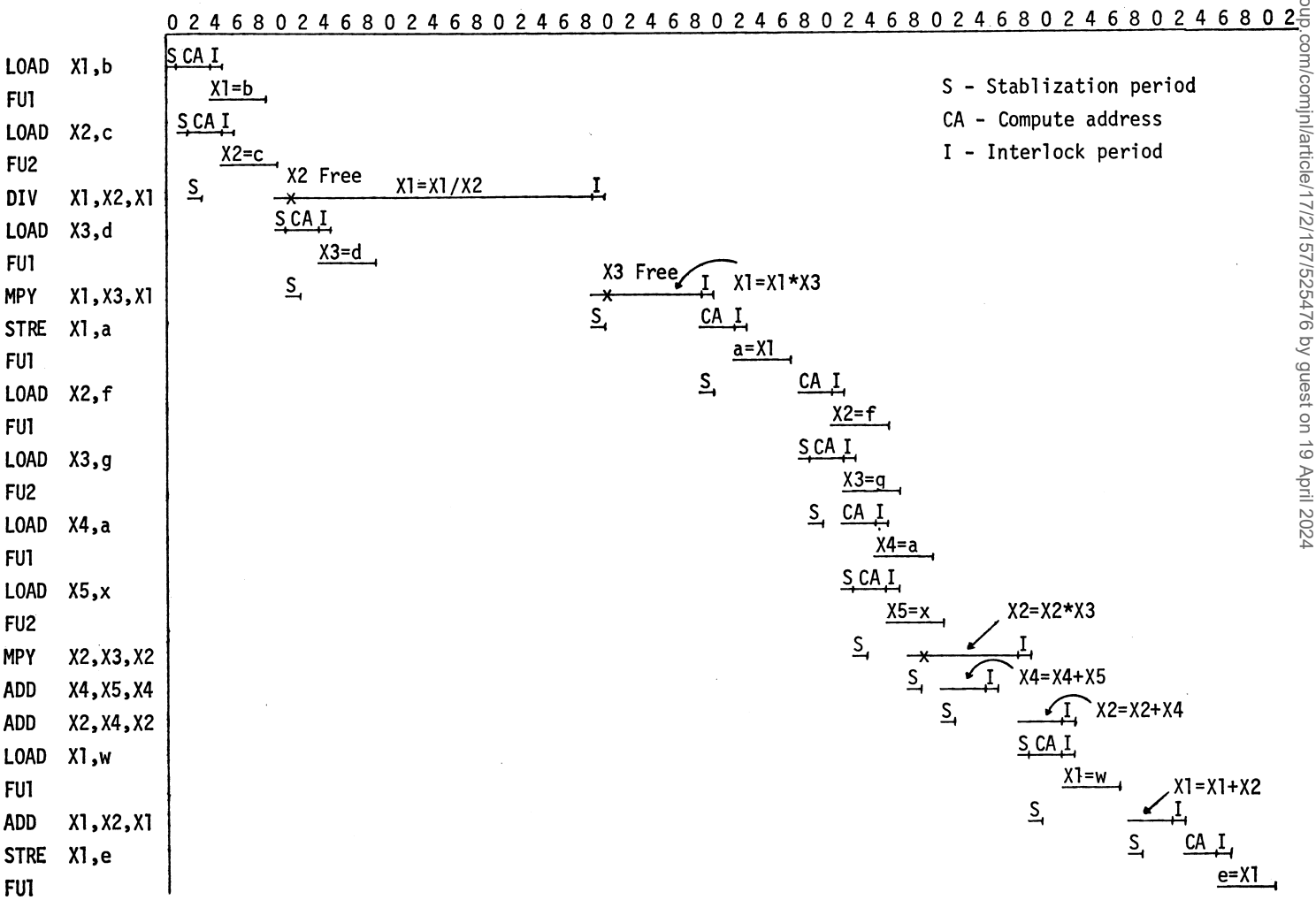


Fig. 2 Execution time chart for the tree in Fig. 1(*b*)

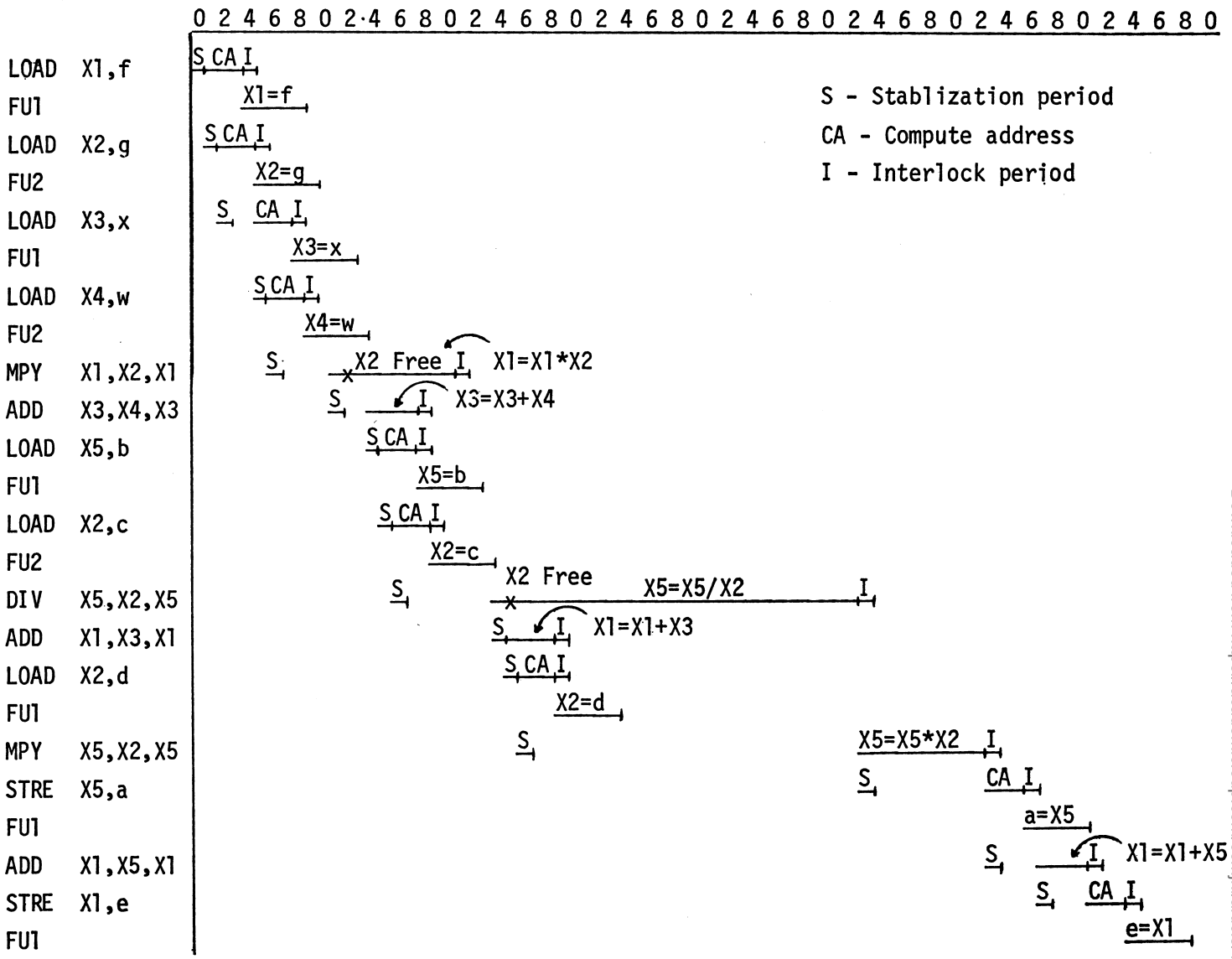


Fig. 3 Execution time chart for the tree in Fig. 1(*c*)

ordered statements.

If we apply the desired variable delaying algorithm mentioned above to the expression in the second statement, we arrive at the tree given in Fig. 1(*c*). In this tree, the operations

$$t_1 = f*g$$
$$t_2 = x + w$$
$$t_3 = t_1 + t_2$$

have been moved up in time (correspondingly, down in the tree) in comparison to the same operations in the tree of Fig. 1(*b*). In **Fig. 3,** a chart is given representing the execution of machine code that corresponds to the tree in Fig. 1(*c*). We see that a total of 79 machine cycles are required to execute the code for the revised statements. Hence, in this instance, the desired variable delaying algorithm would result in a 22% savings of machine cycles for our pseudo 6600.

3. Algorithm for variable delay

The algorithm described below accepts an arithmetic expression and a set of variables to be delayed as inputs. The expression is scanned from right to left and temporary results are generated and an output string is produced. The output string becomes a new input expression and is scanned in a like manner, producing more temporary results and another output string. This process is repeated until a string is scanned and no temporary results are generated. The final output string is the desired reduced expression, and the temporary results produced represent operations that may proceed before the necessary involvement of the delayed variables.

The algorithm assumes that the input expression is composed of single character scalar variable names and the operations listed in **Table 1.** Table 1 also gives each operator's precedence. A list of variables used in the algorithm along with a description of their meaning is given in **Table 2.** A scan operation is a procedure for assigning values from the input string to three variables used in the algorithm—LSCOP, ITEM and RSCOP. When a scan operation is performed, RSCOP is assigned the current value of LSCOP and the next two symbols in the input string from right to left are read. The first symbol is assigned to ITEM and the second symbol is assigned to LSCOP. The third word of STACKS enabled the handling of expressions containing parenthesis. Each time a step up in precedence occurs as a result of a scan operation, the current output string address is stored in the third word of the top entry in the STACKS (if there is one). Then if this entry in the STACKS must be unstacked into the output string, it is added using the address in its third word. If an entry in STACKS must be unstacked into the output string and its third word does not contain a saved output string address, it is added adjacent to the last entry made into the output string. Also, whenever a left and right parenthesis are separated only by an operand in the output string, they are both deleted. The algorithm for variable delay in an expression is as follows.

Step 1. Set K and OLDK to zero and put a FENCE at the start of the input expression.

Step 2. Set LSCOP to '+', out J to zero, and prepare to scan a new input string.

Step 3. Perform a scan operation and

1. if a right parenthesis is to be assigned to ITEM, then if J is not zero, store the current output string address in STACKS (J, 3). Increase J by one and set STACKS (J, 2) to a right parenthesis. Also add a right parenthesis to the output string. Perform a new scan operation by setting ITEM to LSCOP, LSCOP to the next symbol in the input string, and RSCOP to a '+'. If ITEM will be assigned a right parenthesis as a result of the new scan operation, then repeat Step 3.1; otherwise go to Step 4.
2. if a left parenthesis is to be assigned to ITEM, then add STACKS(J, 1) and STACKS(J, 2) to the output string using the address in STACKS(J, 3), decreasing J by one, as long as STACKS(J, 2) is not a right parenthesis. When it is a right parenthesis, decrease J by one and the last symbol entered in the output string is a unary operator. If it is a '+', write a left parenthesis over it, then scan the next symbol in the input string into LSCOP and go to Step 8. If it is a '−', add a left parenthesis to the output string and set a switch to indicate that a unary minus is in the output string. Then scan the next symbol in the input string into LSCOP and go to Step 8.
3. if a FENCE is to be assigned to ITEM, then empty the STACKS into the output string. The last symbol entered into the output string is a unary operator. If it is a '+', overwrite it with a FENCE and go to Step 9. If it is a '−', add a FENCE to the output string and set a switch to indicate that a unary minus is in the output string and go to Step 9.

Step 4. If ITEM contains a variable to be delayed, then

1. if LSCOP and RSCOP equal '**', then if STACKS(J, 2) equals '**', add STACKS(J, 1) and STACKS(J, 2) to the output string and decrease J by one, otherwise just add ITEM and LSCOP to the output string and go to Step 3.
2. if LSCOP is greater than or equal in precedence to RSCOP, add ITEM and LSCOP to the output string and go to Step 3.
3. if LSCOP is less in precedence than RSCOP and J is not zero, add STACKS(J, 1) and STACKS(J, 2) to the output string (decreasing J by one) as long as the precedence of STACKS(J, 2) is equal to that of RSCOP. Then add ITEM to the output string and go to Step 8. When J equals zero, add ITEM to the output string and go to Step 8.

Step 5. If LSCOP is greater in precedence than RSCOP, then if J is not zero set STACKS(J, 3) to the current output string address, otherwise, increase J by one and set STACKS(J, 1) to ITEM and STACKS(J, 2) to LSCOP and go to Step 3.

Step 6. If J equals zero, then

1. if LSCOP is less in precedence than RSCOP, add ITEM to the output string and go to Step 8.
2. if LSCOP and RSCOP equal '**', add ITEM and LSCOP to the output string and go to Step 3.
3. If LSCOP equals RSCOP in precedence, increase J by one and set STACKS(J, 1) to ITEM and STACKS(J, 2) to LSCOP and go to Step 3.

Step 7. If J is not equal to zero, then

1. if RSCOP is equal in precedence to STACKS(J, 2), then increase K by one and
   (*a*) if LSCOP equals '−' and STACKS(J, 2) equal '+', generate
   $$T_k = \text{STACKS}(J, 1) - \text{ITEM}$$
   and set LSCOP to '+'.

Table 1 Allowable operators and their precedences

OPERATORS	PRECEDENCES
Parenthesis (,)	1
Fence |—	1
Add, Subtract +, −	2
Multiply, Divide *, /	3
Exponentation **	4

Table 2 Algorithm variables and their meanings

VARIABLE	MEANING AND USE
FENCE	A unique symbol marking the start of the expression.
ITEM	An operand produced by a scan operation.
J	Index for STACKS.
K	Temporary result counter.
LSCOP	Left operator produced by a scan operation.
OLDK	Count of temporaries previously generated.
RSCOP	Right operator produced by a scan operation.
STACKS	A triple word stack for holding operands, operators and output string addresses.

   (*b*) if LSCOP and STACKS(J, 2) equal '−', generate
   $$T_k = \text{ITEM} + \text{STACKS}(J, 1)$$
   (*c*) if LSCOP equals '/' and STACKS(J, 2) equal '*', generate
   $$T_k = \text{STACKS}(J, 1)/\text{ITEM}$$
   and set LSCOP to '*'.
   (*d*) if LSCOP and STACKS(J, 2) equal '/', generate
   $$T_k = \text{ITEM} * \text{STACKS}(J, 1)$$
   (*e*) otherwise, generate a temporary result of the form
   $$T_k = \text{ITEM STACKS}(J, 2) \text{ STACKS}(J, 1)$$
   In any of the cases above after the T_k is generated, decrease J by one, add the T_k to the output string and go to Step 8.
2. if LSCOP and RSCOP equal '**', add ITEM and LSCOP to the output string and go to Step 3.
3. if LSCOP and RSCOP are equal in precedence, increase J by one and set STACKS(J, 1) to ITEM and STACKS(J, 2) to LSCOP and go to Step 3.
4. if RSCOP is not equal in precedence to STACKS(J, 2), add ITEM to the output string and go to Step 8.

Step 8. If LSCOP is not the FENCE or a left parenthesis, add LSCOP to the output string and go to Step 3. Otherwise,

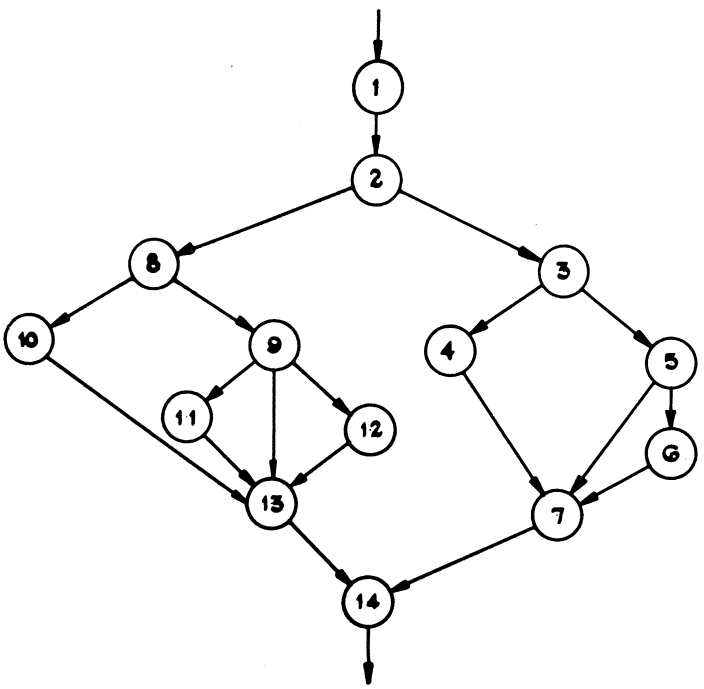
1. if LSCOP is the FENCE, then
   (*a*) if J is zero, add LSCOP to the output string and go to Step 9.
   (*b*) if J is not zero, unstack STACKS(J, 1) and STACKS(J, 2) until J is zero. Then add LSCOP to the output string and go to Step 9.
2. if LSCOP is a left parenthesis, then add STACKS(J, 1) anp STACKS(J, 2) using the address in STACKS(J, 3) to the output string (decreasing J by one) as long as STACKS(J, 2) is not equal to a right parenthesis. When STACKS(J, 2) equals a right parenthesis, simply decrease J by one and add a left parenthesis to the output string. Scan the next symbol in the input string into LSCOP and go to Step 8.

Step 9.

1. If K is greater than OLDK and the switch indicating whether or not a unary minus is in the output string is off, then set OLDK to K and go to Step 2. If the unary minus switch is on and K is greater than OLDK, turn the switch off and

Table 3 Results of one pass of the algorithm $a + b + c * d + e$

LSCOP	ITEM	RSCOP	J	STACKS(J, 1)	STACKS(J, 2)	TEMPS	OUTPUT STRING
+	∅	∅	0	∅	∅		∅
+	e	+	1	e	+		∅
*	d	+	2	d	*		∅
				e	+		
+	c	*	1	e	+	$T_1 = c * d$	$+ T_1$
+	b	+	1	e	+		$+ b + T_1$
|—	a	+	0	∅	∅	$T_2 = a + e$	$\|— T_2 + b + T_1$



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 4 PPTG and its connectivity matrix for the parallel processable program

invoke the unary minus handler. On its return, set OLDK to K and go to Step 2.

2. If K equals OLDK and the unary minus indicator switch is off, the output string contains the desired reduced expression. If this switch is on, turn it off and invoke the unary minus handler. On its return, go to Step 9.

The unary minus handler mentioned in Step 9 of the algorithm locates unary minus operators in the output string and attempts to generate a T_k using one of them. Its general logic is to scan the output string for a unary minus. When one is found, the code to the right of the unary is scanned left to right in an attempt to locate an operand which the unary can be applied to. If one is found, K is increased by one and a T_k is generated of the form:

$$T_k = -\text{operand}$$

The output string is adjusted by deleting the unary minus, and the operand used is replaced by the T_k. Then a return occurs. If the operand found is itself a T_k where K is greater than OLDK, it is not considered since we want the algorithm to generate only parallel processable temporaries in a given pass.

If the unary cannot be applied, the scan is resumed in search of another unary minus. If one is not found, a return occurs.

In **Table 3,** results are presented for the algorithm's application to the example expression $E = a + b + c * d + e$ for one pass, where b is the variable being delayed. Notice that all temporary results generated for a particular pass of the algorithm are independent of one another, and can therefore be executed in parallel. This is a desirable by-product of the algorithm.

The algorithm has been coded in FORTRAN and the program has been tested using numerous examples. The desired results were produced in each case. A copy of the program is available upon request from the author.

4. Method for using the variable delay algorithm

The most widely publicised technique for global recognition of parallel processable code in computer programs is that of Ramamoorthy and Gonzalez (1969*a*, *b*). Basically, their method is as follows. A FORTRAN source program is input, and a sequential graph model of the program is developed. A reduced graph is formed by loop reduction and then based on inter-statement data dependencies and any essential ordering among the statements, the reduced program graph is transformed into a parallel graph model. From this graph, all parallel processable tasks in the program and the sequence in which they must be executed is derived. An example parallel program task graph (PPTG) and its connectivity matrix for a hypothetical program are given in **Fig. 4.**

To incorporate the variable delay algorithm into Ramamoorthy and Gonzalez's scheme, some additional information about the source program must be gathered. During the first phase of their method as the source program is read in and program tasks are identified (a statement corresponds to a task), a list L of tasks numbers should be constructed so that if $k \in L$, then program task t_k contains an arithmetic expression composed of two or more operations.

For program task t_i let I_i and 0_i denote the input/output sets respectively of t_i. Furthermore, it is assumed that task numbers correspond to graph node numbers such that row i and column i in a connectivity matrix correspond to task t_i.

After Ramamoorthy and Gonzalez's technique has been applied and the parallel program task graph (PPTG) has been derived for each loop in the program and the reduced program itself, then the following steps are necessary to apply the variable delay method. Let PPTG* denote the transitive closure of PPTG (there will be a PPTG* for each loop in the program as well as the reduced program). PPTG* can be formed from

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1	1	1	1	1	1
3	0	0	0	1	1	1	1	0	0	0	0	0	0	1
4	0	0	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	0	0	1	1	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	0	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	1	1
11	0	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	1	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(*a*)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(*b*)

Fig. 5 PPTG* and the matrix A for the example parallel processable program

PPTG using Warshall's (1962) algorithm. PPTG* for the example PPTG of Fig. 4 is given in **Fig. 5**(*a*).

For each $k \in L$, identify the set P_k of task numbers that correspond to the tasks which are predecessors of t_k. P_k is formed very simply using the appropriate PPTG*. It is the set of row indexes of PPTG* so that $j \in P_k$ implies $\text{PPTG}^*_{jk} = 1$. For example, suppose task t_{13} in Fig. 4 contains an arithmetic expression of two or more operations. Then from PPTG* given in Fig. 5(*a*) we have

$$P_{13} = \{1, 2, 8, 9, 10, 11, 12\} .$$

For each $k \in L$, also identify the set P'_k of task numbers that correspond to the tasks that are immediate predecessors of t_k. A task t_i, which is not an unconditional branch vertex, is considered to be an immediate predecessor of t_k if $\text{PPTG}_{ik} = 1$ and there does not exist a task t_j such that $\text{PPTG}_{ij} = 1$ and $\text{PPTG}_{jk} = 1$. P'_k is formed using a technique developed by Stevens (1971) for removing redundant arcs from a graph. Let

$$A = \text{PPTG}^* \cap \sim(\text{PPTG}^*)^2 ,$$

then P'_k is formed using the matrix A. It is the set of row indexes of A such that $j \in P'_k$ implies $A_{jk} = 1$. In Fig. 5(*b*), the matrix A for the example PPTG* is given, and from it, for task t_{13}, we have

$$P'_{13} = \{10, 11, 12\} .$$

Before proceeding further it should be noted that if there exists $j \in P'_k$ such that task t_j is a decision vertex, then the attempt to apply the variable delay algorithm to the expression in task t_k should be abandoned. Since t_j is a decision vertex, this implies that t_k is a conditional successor of t_j. Hence, any computations removed from t_k would still have to wait on the completion of task t_j before they could be initiated. Thus nothing would be gained by the application of the variable delay algorithm. However, this restriction does not hold if the scheme of processing successors of conditionals in parallel is being used. In this scheme, the various paths emanating from a decision vertex are worked on in parallel with the decision vertex. When its successor path is finally determined, the work done on the other paths up to that point is abandoned and processing continues along the successor path.

Suppose P'_k has m_k elements. Let

$$U = \bigcup_{i=1}^{m_k} 0_{p_i}$$

where $p_i \in P'_k$ for each i. The variables that should be delayed in the expression in task t_k are the elements of the set D, where

$$D = U \cap I_k .$$

Therefore, the set D and the expression E in task t_k would be passed to the variable delay algorithm. All computations in E that can proceed prior to the necessary involvement of the variables in D would be identified. If these computations are collected into a new task t'_k, and if we replace E in t_k by E', the reduced expression generated by the variable delay algorithm, then t'_k can be initiated in parallel with any or all of the tasks represented in P'_k.

The predecessors of t'_k in the graph are the tasks in the set P''_k, where

$$P''_k = P_k - P'_k .$$

('$-$' denotes relative compliment). The only successor task of t'_k is t_k itself. Notice that if no temporary results are generated by the variable delay algorithm, then we have no new task t'_k. The implication here is that the variable delay method cannot improve the parallel processability of the program as far as task t_k is concerned.

Assuming there are some temporary results generated by the algorithm, then t'_k must be added to the appropriate PPTG by adding an additional row and column to the connectivity matrix representing the PPTG. This will yield an augmented connectivity matrix A(PPTG). If we let k' denote the task number corresponding to t'_k, then set $A(\text{PPTG})_{k'k} = 1$ and set $A(\text{PPTG})_{jk'} = 1$ for each $j \in P''_k$. Elsewhere in the k'th row and column of A(PPTG) there is a zero.

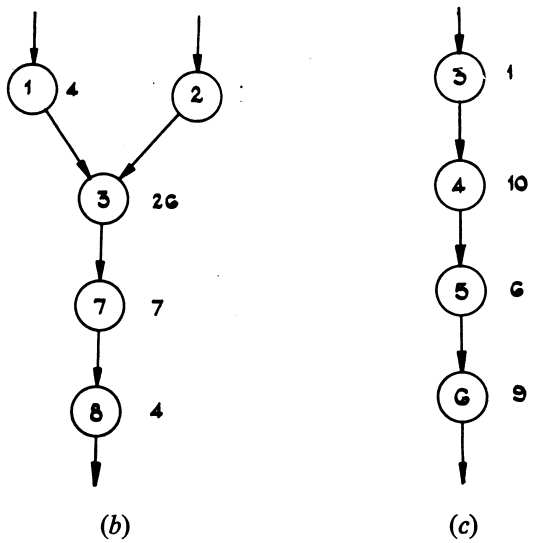
After each task t_k, $k \in L$, has been processed by the method just described, the technique of 'precedence partitions' (Ramamoorthy and Gonzalez, 1969*a*, *b*) can be applied to the final A(PPTG) which will yield the parallel processable tasks for the program and the sequence in which they must be executed. Therefore, the variable delay method can be incorporated into Ramamoorthy and Gonzalez's technique very easily. PPTG* and the matrix A need be formed only once and are used to process each task represented in L. Also, the only additional information needed about the source program from that already collected by their technique is what tasks contain expressions of two or more operations. There is no problem in adding additional rows and columns to PPTG to form an augmented matrix. Hence, unlike Volansky's (1970) method for discovering hidden parallelism, the variable delay method is comparatively simple to employ.

5. Example application of the variable delay method

Using the variable delay algorithm and the method described for incorporating it into Ramamoorthy and Gonzalez's (1969*a*, *b*) technique, an example program will be analysed. This will illustrate the method of variable delay, and demonstrate how the technique can be beneficial by recognising

```
1       FUNCTION CHEBY(A, B, M,F)
        IMPLICIT REAL*8 (A-H, O-Z)
        REAL*8 CHEBY, A, B, F
2       SUM = 0.
3       DO 1 I = 1, M
4       Z1 = DCOS(FLOAT(2*(I - 1) + 1)*3.1415927/
          FLOAT(*2M))
5       X1 = (Z1*(B - A) + B + A)/2.
6     1 SUM = SUM + F(X1)*DSQRT(1. - Z1*Z1)
7       CHEBY = (B - A)*3.1415927*SUM/FLOAT(2*M)
8       RETURN
        END
```

(*a*)



(*b*) (*c*)

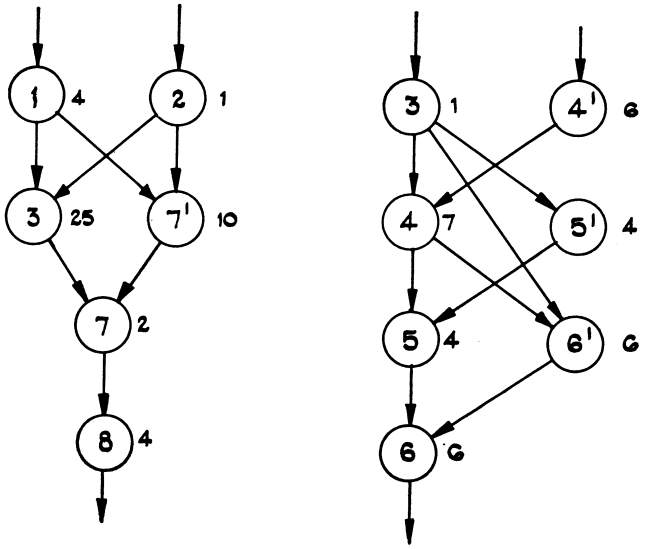
Fig. 6 Example program for the variable delay method

additional parallel processable code in computer programs.

To illustrate the application of the variable delay method, a FORTRAN function subprogram CHEBY for computing the value of the integral of $F(X) * D(X)$ between the integration limits A and B using the M-point Gauss-Chebyshev quadrature formula will be used. The complete function CHEBY as it appears in Carnahan, Luther and Wilkes (1969) is given in **Fig. 6**(*a*).

The parallel program task graphs which resulted from applying Ramamoorthy and Gonzalez's technique are given in Fig. 6(*b*) and (*c*) for the reduced program and the single DO-loop in the program respectively. The nodal numbers in the two graphs correspond to the executable statement numbers assigned to the executable statements in the program. The integer beside each node in the graphs represents the total number of operations (either implied or specified) to be performed in the statement corresponding to that node. Function references are treated as single operations. The total number of sequential operations to be performed in a graph is defined as $\sum_{i=1}^{p} m_i$ where p is the number of precedence partitions for the graph (Ramamoorthy and Gonzalez, 1969*a*, *b*) and m_i is the number of operations to be performed at node n in precedence partition i and node n has the maximum number of operations of all the nodes in precedence partition i. Hence, the total number of sequential operations to be performed in the graph of Fig. 6(*b*) is 41, and for the graph in Fig. 6(*c*) there are 26 total sequential operations.

Applying the variable delay method detailed in the previous sections, we arrive at the two augmented parallel program task graphs given in **Fig. 7**(*a*) and (*b*). Again, the integer beside each node represents the total number of operations to be performed at that node. The new and revised tasks that resulted from applying the variable delay algorithm are given in **Table 4.** In



(*a*) (*b*)

Precedence Partitions
{1, 2}
{3, 7'}
{7}
{8}

(*c*)

Precedence Partitions
{3, 4'}
{4, 5'}
{5, 6'}
{6}

(*d*)

Fig. 7 Augmented parallel program task graphs and their precedence partitions for the example program

Fig. 7(*c*) and (*d*) the precedence partitions indicating what tasks in each augmented graph can be processed in parallel and the sequence in which they must be executed are given.

Notice that the total number of sequential operations to be performed in the graph of Fig. 7(*a*) is 35 and that of Fig. 7(*b*) is 25. Comparing the total number of sequential operations, we see that the variable delay method reduced this number by 15% for the graph in Fig. 6(*b*), and by 4% for the graph in Fig. 6(*c*). This represents a significant improvement in the parallel processable code exposed by just using Ramamoorthy and Gonzalez's technique alone. Equating the total number of sequential operations to be performed with the time required to execute the program represented by a graph, this would also represent a significant decrease in program execution time. Also, the 4% savings for the loop only represents that for one iteration.

Table 4 Results of the variable delay method on function CHEBY

NEW TASK NUMBERS	TEMPORARY RESULTS	REVISED OLD TASK
4'	T41 = 2*M	Z1 = DCOS(FLOAT
	T42 = FLOAT(T41)	(2*(I − 1) + 1)*T43)
	T43 = 3.1415927/T42	
5'	T51 = B − A	X1 = (Z1*T51 + T52)/2.
	T52 = B + A	
6'	T61 = Z1*Z1	1 SUM = SUM + F
	T62 = 1.−T61	(X1)*T63
	T63 = DSQRT(T62)	
7'	T71 = 2*M	CHEBY = SUM*T75
	T72 = B − A	
	T73 = FLOAT(T71)	
	T74 = T72*3.1415927	
	T75 = T74/T73	

Since the loop is executed M times, the saving becomes even more significant.

6. Conclusions

An algorithm for variable delay in an arithmetic expression has been presented, and its use in a scheme for recognising parallel processable code in computer programs has been described. The implications of the variable delay method and its apparent value of discovering hidden parallelism have been illustrated through analysis of an example program.

The algorithm in its present form will not handle expressions that contain array or function references. However, only minor modifications are necessary to incorporate this feature. This and the value of the method will be the object of further research in this area.

Acknowledgement

The author expresses his gratitude to Professor C. V. Ramamoorthy and his colleagues for their encouragement and assistance.

References

BAER, J. L. (1972). *A Survey on Multiprocessing*, TR-72-05-01, Computer Science Group, University of Washington, Seattle, Washington.

BERNSTEIN, A. J. (1966). Analysis of Programs for Parallel Processing, *IEEE Trans.*, EC-15, pp. 757-763.

BINGHAM, H. W., FISHER, D. A., and REIGEL, E. W. (1967). *Automatic Detection of Parallelism in Computer Programs*, TR-ECOM-02463-4, Burroughs Corp., TR-67-4, AD 622 274.

CARNAHAN, B., LUTHER, H. A., and WILKES, J. O. (1969). *Applied Numerical Methods*. New York: John Wiley & Sons, Inc.

CONTROL DATA 6400/6500/6600 Computer Systems Reference Manual (1969). Control Data Corporation, St. Paul, Minnesota.

LORIN, H. (1972). *Parallelism in Hardware and Software: Real and Apparent Concurrency*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

STEVENS, W. L. (1971). *An Automated Method for the Editing, Construction and Graphical Display of Activity Networks*. (Ph.D. Thesis), Texas A. & M. University, Department of Industrial Engineering, College Station, Texas.

RAMAMOORTHY, C. V., and GONZALEZ, M. J. (1969*a*). *Recognition and Representation of Parallel Processable Code in Computer Programs*, N71-11333, The University of Texas, Austin, Texas.

RAMAMOORTHY, C. V., and GONZALEZ, M. J. (1969*b*). Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs, *Proc. FJCC, AFIPS*, Vol. 35, pp. 1-15.

RUSSELL, E. C., and BAER, J. L. (1970). Preparation and Evaluation of Computer Programs for Parallel Processing Systems, in Hobbs, L. C., Thesis, D. J., Titus, H., Trimble, J. and Highberg, I. (Eds.), *Parallel Processor Systems, Technologies, and Applications*. New York: Spartan Books, pp. 375-416.

VOLANSKY, S. A. (1970). Graph Model Analysis and Implementation of Computational Sequences. (Ph.D. Thesis), University of California at Los Angeles, Department of Computer Science.

WARSHALL, S. (1962). A Theorem on Boolean Matrices, *JACM*, Vol. 9, pp. 11-12.

Book reviews

Computational Methods in Ordinary Differential Equations, by J. D. Lambert, 1972; 278 pages. (*John Wiley and Sons Ltd.*, £5·50)

The title is slightly misleading, as the book is almost entirely concerned with initial-value problems; there are just five pages at the end concerned with the boundary-value case. Dr. Lambert takes a middle road between the rocky highlands of abstract numerical analysis and the muddy ground of routine computation, where the algorithms grow. As he states in his preface, few theorems are stated and even fewer are proved, but there is a great deal of careful analysis, and references are given where necessary.

A large part of the book is concerned with a careful discussion of linear multi-step methods, both from the theoretical and the practical standpoint. This section contains an admirably thorough treatment of the various types of instability which may arise. There follows a chapter dealing with Runge-Kutta methods, leading on to discussions of hybrid formulae and extrapolation methods. The whole of this part of the book deals only with the solution of a single first-order differential equation.

Chapter 8 then extends the methods to deal with systems of simultaneous equations, and in particular to stiff systems. This chapter gives the impression of being rather hurried, especially by comparison with the detailed arguments of the earlier part. Here the author does not seem to me to give sufficient weight to the practical problems which arise. In practice a single equation does not appear very often. Systems of two or three equations are more usual, and twenty or more are all too common. The very practical problem of implementing an implicit method for such a system is dismissed rather briefly in this chapter. The discussion of stiff systems is also rather brief, but this is a field which is still being vigorously tilled, and Dr. Lambert mentions a number of recent advances.

The publication of this book follows closely on the appearance of C. W. Gear's *Numerical Initial Value Problems in Ordinary Differential Equations*.* The two books cover so closely the same ground that a review will involve a comparison, which is worth making explicit. Both bring the reader up to date with recent work; both have good bibliographies, and it is interesting that each of them has a substantial number of references which do not appear in the other. Gear gives a much more formal treatment, in the form of lemmas, theorems and proofs, where Lambert gives a discussion with references. At the other end of the spectrum, Gear gives some complete FORTRAN programs, and thus tries to cater for the practical user of algorithms as well as the pure numerical analyst. Lambert, however, keeps to the middle road, and seems to me much more successful in providing the practical man with the explanation and information which he needs. In particular, Lambert gives more than just a survey of the vast collection of methods, with their derivation. When he believes that Method A is better than Method B, he is quite ready to say so.

D. F. MAYERS (Oxford)

*Reviewed in this *Journal* (Volume 15, number 2, May 1972, page 155).

Computer Science: Projects and Study Problems, by Alexandra I. Forsythe, Elliott I. Organick, and Robert P. Plummer, 1973; 292 pages. (*John Wiley and Sons Ltd.*, £2·85)

This book is a companion to the main texts and language supplements of a series that has developed over the last few years. This latest offering contains the specification of thirteen programming projects with hints and background discussion. These are of varying complexity and between them offer a thorough grounding in the practice of the programming art. In addition there are a large number of smaller exercises with examples and discussion on more detailed topics.

The major flaw in this otherwise valuable collection is the continual cross-referencing to the main texts—this being most pronounced in the study problems which are linked by chapter and section. This makes it less useful to those who are not hooked on the series, which is a pity. Rather worse, it gives those who are hooked one more excuse for not looking at the world outside. Most parents will be familiar with the problems of weaning their children away from the Famous Five, and I greatly fear that Alexandra I. Forsythe (who seems the most plausible candidate) is set to become the Enid Blyton of the programming schools.

C. M. REEVES (Leeds)