Semi-Automatic Allocation of Data Storage for PACT I*

J. I. DERR

The RAND Corporation, Santa Monica, Calif.

AND

R. C. LUKE

Lockheed Aircraft Corp., Burbank, Calif.

Introduction

One of the principal objectives of the PACT I compiler has been to eliminate as much as was immediately feasible of the bookkeeping and rudimentary thinking which is involved in the preparation of a computational problem for a large-scale, high-speed digital computer.[1] To this end, a set of PACT pseudo-operations was developed. These operations are more closely related to the computational problem than the machine operations. In addition a pseudo-language was devised for addressing operands. The function of PACT is to compile a machine-language code from a given PACT pseudo-language code. In this sense PACT shortens the gap between the computational model and the final machine code by performing much of the intervening work. In this paper we shall be primarily concerned with the system which was designed to allocate machine storage locations for the operands which are involved in PACT pseudo-operations. The system was made to automatically allocate storage locations up to the total capacity of the high-speed memory and to allow the user some convenient means for constraining this allocation when necessary. No provision was made for automatic secondary storage allocation.

General Discussion of Storage Assignment

Several general types of storage can be classified in order to facilitate the following discussion. Under this enumeration we single out temporary storage, number storage, variable storage, and instruction storage.

By *temporary* (working) storage, we mean the storage which is used primarily to retain the intermediate arithmetic results of PACT instruction steps which are referred to in other PACT I instructions by their step numbers. In Figure 1 the result of step 2 will be placed in temporary storage for use in step 5. Any such results which need to be retained within a region are automatically assigned to temporary storage by the compiler.

The *numbers* are the data which are written explicitly as operands in relevant PACT instruction steps. The number 123 in Figure 1 will be automatically

* Presented at the meeting of the Association, Sept. 14–16, 1955.

[1] The reader should assume throughout the paper that the computer is the IBM Type 701.

PACT I Code Sheet

REG	STEP	OP	FACT	S_1	S_2	Q	N
ABC	1		123				
	2.	−	A				
	3	+	B				
	4	EQ	C			7	
	5		R 2				

FIG. 1

assigned to number storage. Optimal use of storage space is effected here by assigning equal numbers to the same memory locations.

The *variables* are the data which are addressable by unique symbolic names and these names are independent of the current values of the data. This invariance arises from the fact that each variable is assigned permanent storage locations, and only the contents of these locations can be altered. In Figure 1, A, B, and C are variables.

The three types of storage already described represent examples of data storage. In contrast there is that storage used to retain the resulting machine-language code; it is called *instruction storage*. PACT I automatically assigns *instruction* storage locations independently of the data storage.

It might be pointed out here that each type of storage is located in an independent block or region of high-speed memory. Since the compiler, at the time these assignments are made, does not know how much storage to set aside for each block, the assignments of these storages are made relative to the beginning of the block. Then just before the final compiled program is ready to be executed, these blocks are assigned to actual memory locations which are usually non-overlapping.

As was stated above, the number, temporary, and instruction storage assignments are made automatically by PACT I. Variable storage assignment, however, is made automatically only under certain limitations, and this type of assignment needs further discussion.

First, variable storage is allocated for vectors and matrices, as well as for scalars, and the dimensions of these arrays may be dependent on some parameters of the problem. In either event, the compiler must be told the maximum extent of each array in order to assign them the correct amount of storage.

Secondly, variable storage—especially when arrays are to be operated on—usually constitutes a very significant portion of storage. By allowing two or more variables to occupy the same storage locations, an extensive optimization of high-speed memory usage can be effected.

There is another problem associated with variable storage. When several sections of a program are to be compiled separately, it is sometimes necessary that the results of one section be accessible to another. The compiler can be signaled to assign the same locations to these common variables.

We are now ready to state the task that was set for the PACT data allocation system. Ordinarily, all storage assignments are to be made automatically. Variable storage assignments, though, will require that there be provided a convenient means for specifying the maximum ranges of arrays and constraints on the assignment where the user deems necessary. The remainder of this paper will be concerned with the allocation of variable storage.

Description of Variable Storage Allocation

Before describing how to specify the maximum range for each dimension of an array, we shall first indicate what PACT I considers to be an array. An array is any set of data which can occupy more than one storage location and is referred to by only one symbolic name. By specifying the range for either dimension of a variable to exceed one, we are defining that variable as an array. The definitions are made by recording the proper information on the variable definition sheet illustrated in Figure 2.

On line one of Figure 2, the variable A is defined to be at most a 5 by 10 matrix. The variable B is defined as a vector with a maximum of 10 elements. Information on this sheet is punched into variable definition cards (one card to a line) and is read by the compiler shortly after the last PACT instruction card has been read.

On the code sheet, elements within an array are indicated by *inductive* subscripts. There is another type of subscript we call *definitive*. A definitive subscript can be considered as an extension beyond the factor field of the name by which we specify the variable. In step 5 of the coding sheet HA is a definitive subscript since ALP has not been defined as an array. In step 4, J is an inductive subscript since B has been defined as a vector.

Any particular values of inductive subscripts must be associated with a particular storage location of an element within an array. This is done by means of the location of the first element and the intervals between storage locations of the elements in each dimension. For matrices, if we call Δ_1 the interval between the location of elements in D_1 and Δ_2 the interval between the location of elements in D_2, then the association is given by equation (1) of Figure 2. Here the location of A_{11} is the first location assigned to the matrix A. Δ_2 is equal[2] to 2 and Δ_1 is equal to 2 D_2. That is, D_1 and D_2 are translated into Δ_1 and Δ_2 for the use of the succeeding sections of the compiler.

A variable will be assigned relative storage locations and scaling if and only if the variable—together with its scaling, dimension information, and constraints—is recorded either on the variable definition sheet or in a PACT instruction step with an EQ operation. Thus, the scalar in step 5 of the coding sheet will be assigned a scaling of 2 as specified on line 3 of the variable definition

[2] Δ_2 is always 2 for the IBM Type 701 since consecutive full-word storage locations are referenced by the consecutive even integers.

PACT I Variable Definition Sheet

FACT	S_1	S_2	Q	D_1	D_2	FACT	S_1	S_2	TYPE	LOC.
A			1	5	10					
B			3	10						
ALP	HA		2							
D			3	5	10					

PACT I Code Sheet

REG	STEP	OP	FACT	S_1	S_2	Q	N
XYZ	1	SET		I	1		
	2	SET		J	1		
	3		A	I	J		
	4	+	B	J			
	5	−	ALP	HA			
	6	EQ	D	I	J		
	7	TEST		J	M		
	8	TEST		I	L		

(1) LOCATION OF A_{IJ} = (LOCATION OF A_{11}) + $(I - 1)\Delta_1 + (J - 1)\Delta_2$

FIG. 2

sheet. Note that although the matrix D occurs in step 6 with an EQ operation, it must be defined as an array on the variable definition sheet.

Assume for the remainder of this section that none of the variables to be assigned have constraints. Then the procedure for assigning variable storage is very straightforward. The variables, beginning with those punched in the variable definition cards, will be assigned consecutive storage locations according to the order in which the variable definition cards are read and then in the order in which the PACT instructions with EQ operations are encountered by the compiler. As a result of this stage of the compiling process, each variable—together with its scaling, dimensions information and first relative location—will occupy a position in a table. This table is called the *variable table.*

In the next stage of the compiling process each PACT instruction is examined for a variable. If a variable is present, it is searched for in the variable table and the information in the table is stored with the instruction. When this stage is completed, a listing of the information in the table is made for the coder's future reference.

As far as coding techniques are concerned, probably the most interesting thing in this section of the compiler is the method of determining the positions of the variables in the variable table. Given the amount of high-speed storage available for the table, we chose a method which would minimize the time required to search for the variables in the table. Since the symbolic name of each variable is formed as a three character base 48 word, only 17 bits are required to retain each name ($48^3 = 27 \cdot 2^{12} < 2^{17}$). Now, the variable table consists of 128 blocks and each block contains four variable positions. In order to determine the block to which a variable will be assigned, form the sum of the first five binary bits, the second six bits, and the remaining six bits of the 17 bit word representing the variables. Next take the resulting sum modulo 128 to be the block into which the variable will be positioned. Since a many-one correspondence exists between variables and blocks, more than one variable could be positioned in the same block. In this event the successive variables will be assigned to the first unoccupied position within a block. If a block already contains four variables, then the variable will be positioned in the first succeeding block which does not contain four variables.

A similar procedure is followed when the variable table is being searched with an argument variable for the corresponding entry variable. With the computed block as the origin, the entry variables in the consecutive variable positions in the table are now compared with the argument variable until the two have identical symbolic names.

Practical experience with the method has confirmed the predictions of the original analysis of the table search problem to the effect that the distribution of the computed block numbers has proved to be sufficiently random for "average sized" problems. For example, if fifty different variables are used in a problem, then the "average" distribution of variables will result in eight blocks containing two variables, one block containing three variables, one block containing four (including the case where more than four variables determine the same

VARIABLE TABLE

BLOCK #62				BLOCK #63				BLOCK #64			
Pos 1	Pos 2	Pos 3	Pos 4	Pos 1	Pos 2	Pos 3	Pos 4	Pos 1	Pos 2	Pos 3	Pos 4
-1S				M - -	-1T						

VARIABLE	BINARY REPRESENTATION	COMPUTED BLOCK #
M - -	001111 110000 000000	63
-1T	000000 000000 111111	63
-1S	000000 000000 111110	62

FIG. 3

computed block number) variables, and the remaining forty blocks containing only one variable.

Constraints on Variable Assignment

In this section we shall describe the methods by which the coder may constrain the allocation of variable storage. One such constraint allows the direct specification of the relative address to be assigned to a variable. On line one of Figure 4, the ten by twenty matrix E is constrained to have its first element located at variable storage location 1000. This constraint could have been used because E had been assigned location 1000 during a previous compilation of another part of the problem. On lines 2 and 3 we have caused F and G to occupy the same part of memory. In case a problem overflows the high-speed memory, the overlapping of variable storage becomes necessary.

The REL constraint allows complete flexibility for specifying assignments. However, if the coder uses the REL constraint for the sole purpose of constraining variable storage to overlap, then he must know or compute the amount of storage for each variable. It was felt that this bookkeeping burden should be assumed by the compiler. Therefore, other more convenient constraints on variable assignment have been devised.

If the user desires to *permit* a variable to occupy the same storage locations as some other variable (called the constraining variable), he can accomplish this by using a SYN (synonym) constraint. On line 2 of Figure 5 the matrix B is constrained to start at the same storage locations as the constraining vector A. (A has been defined as a vector on line 1.) Since A and B are allowed to overlap, they must be used in different phases of the problem. As a result of the SYN constraint B is automatically assigned in a phase one greater than that of the constraining variable A. Below the variable definition sheet in Figure 5 is a line diagram of the assignments made as a result of the constraints specified. The lines represent consecutive relative locations. Here, both B and C have been specified to overlap A. Since A is a smaller array than both B and C, and B has been read by the compiler before C, only B will be assigned the same first location as A. The SYN constraint only permits the possibility of overlapping.

In Figure 6 we have the same example except that here C is constrained to overlap B instead of A. If the coder desires to have the storage locations allo-

PACT I Variable Definition Sheet

FAC	S_1	S_2	Q	D_1	D_2	FAC	S_1	S_2	TYPE	LOC.
E			4	10	20				REL	1000
F			7	5	10				REL	200
G			8	5	10				REL	200

FIG. 4

PACT I Variable Definition Sheet

VARIABLE						CONSTRAINT				
FAC	S_1	S_2	Q	D_1	D_2	FAC	S_1	S_2	TYPE	LOC
A			1		10				REL	0
B			3	5	5	A			SYN	
C			0	5	5	A			SYN	
D			2	5						

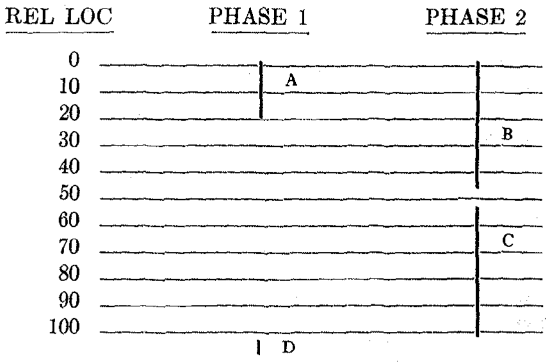

FIG. 5

cated to a variable follow—that is, be larger numerically than the locations allocated to another variable regardless of the order of the variable definition cards—he can effect this by using a SUC (Succeeds) constraint. This constraint insures that the variables will be in the same phase and can be used for this purpose alone where the ordering is not important. Figure 7 illustrates the assignments made with a succeeds constraint. D has not been overlapped here. If this is desired, an IMS (Immediately Succeeds) constraint can be specified. This constraint insures that the storage assigned to the variable will always follow immediately after the storage assigned to the constraining variable. This point is illustrated in Figure 8.

These four constraints, REL, SYN, SUC, and IMS—together with the order of the variable definition cards—constitute all the means by which the coder may control variable storage assignments. The execution of these constraints by the compiler greatly complicates the previously described procedure for assigning variable storage. The modified procedure will now be discussed.

To facilitate the following description we shall first define the term *chain.* All variables whose assignments depend (dependence is a transitive relation) on the assignment of a generating variable (that is, by means of constraints) will constitute a chain of that variable. For example, in Figures 7 and 8 the variables A, B, and D form chains of the variable A. A is said to generate the chains.

PACT I Variable Definition Sheet

VARIABLE						CONSTRAINT				
FAC	S_1	S_2	Q	D_1	D_2	FAC	S_1	S_2	TYPE	LOC
A			1	10					REL	0
B			3	5	5	A			SYN	
C			0	5	5	B			SYN	
D			2	5						

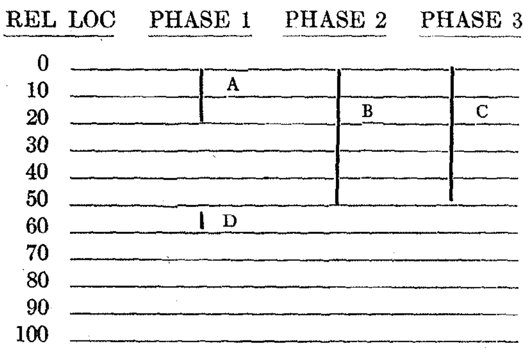

Fig. 6

PACT I Variable Definition Sheet

VARIABLE						CONSTRAINT				
FAC	S_1	S_2	Q	D_1	D_2	FAC	S_1	S_2	TYPE	LOC
D			4		5	A			SUC	
B			6	5	5	A			SYN	
A			3		10					

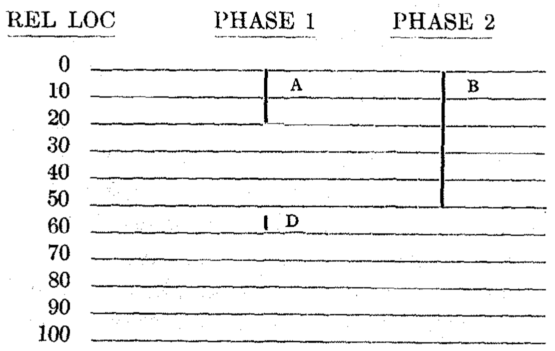

Fig. 7

PACT I Variable Definition Sheet

VARIABLE						CONSTRAINT				
FAC	S_1	S_2	Q	D_1	D_2	FAC	S_1	S_2	TYPE	LOC
D			4		5	A			IMS	
B			6	5	5	A			SYN	
A			3	10						

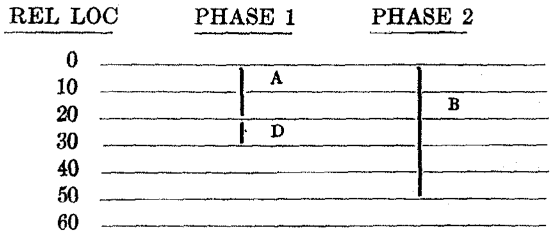

Fig. 8

On the first level the allocation of storage is made by chains. Each variable belongs to a unique chain which might consist of just the variable itself. The order in which the chains are allocated storage essentially depends first upon the order in which the variable definition cards are read and secondly, upon the order in which the PACT instruction steps are investigated when searching for variables defined by EQ instructions. The normal procedure is to assign the first chain starting at relative location zero and assign each succeeding chain starting immediately after the preceding one.

However, the allocation of storage for chains of variables generated by REL constrained variables will always take precedence over the storage allocation for chains generated by non-constrained variables. In addition the chains dependent on non-constrained variables are not allowed to overlap the chains dependent on REL constrained variables. In Figure 5, for example, A, B, and C form a chain of the REL constrained variable A. D is a non-constrained variable and therefore is not allowed to overlap the foregoing chain.

On the second level we can consider how the allocation is made within individual chains. The variable which generates the chain is assigned first and the phase number is one. After *any* given variable has been assigned, the following process takes place:

1. Search among all of the SYN constrained variables which have not yet been assigned for the first one which has the constraint variable exactly the same as the given variable.
   a. If such a variable exists, increase the phase number by one and assign the variable so as not to overlap any variable in the chain which has been assigned in a higher phase. Repeat Step 1.
   b. If no such variable exists, proceed to Step 2.

2. Search among all of the SUC and IMS constrained variables which have not yet been assigned for the first one which has the constraint variable exactly the same as the given variable.
   a. If such a SUC constrained variable exists, assign the variable so as not to overlap any variable in the chain which has been assigned in a *higher* phase. If such an IMS constrained variable exists, assign the variable immediately after the given variable. Repeat Step 1.
   b. If no such variable exists, proceed to Step 3.
3. a. If the phase number is one, proceed to the next chain.
   b. If the phase number exceeds one, decrease the phase number by one. Repeat Step 1.

Conclusion

The choice of this particular system for allocating data storage was greatly influenced by the scope of the PACT I project and the structure of the PACT working committee. In order to distribute the work involved in producing the compiler among numerous installations, the compiler was broken down into several sections, one of which was data storage allocation. This partitioning made it desirable to make the data storage allocation as independent as possible from the allocation of the program storage. However, in order to make an optimum automatic allocation of storage, it would have been necessary to closely coordinate the data storage allocation section with the sections that expand the program code, which would have meant that the scope of PACT I would have had to be enlarged.

In order to fix the idea, let us consider some of the additional things the compiler would be required to do in order to impose automatically the SYN, SUC, and IMS constraints, make secondary storage allocation, and to do this in some optimum fashion. The compiler would, of course, only make an optimum allocation based upon the rules built into the system. Now, if we were to neglect the efficiency factor, the task accorded to the compiler would be relatively easy. The transmission of data between the high-speed storage and the auxiliary storages could be accomplished according to some relatively simple scheme. But in order to do this efficiently, the compiler would need to make additional judgments concerning the frequencies with which groups of operations are performed, input-output to computing ratios, etc. Essentially, the compiler would be required to follow the logic of the entire program.

In order to construct a system which will automatically and efficiently make auxiliary storage assignments and provide for the overlapping of high-speed storage for both the program and the data, much more effort will have to be expended than was for PACT I. The increase in difficulty is analogous to that involved in the application of Boolean algebra to the design of electrical systems involving "sequential" (feedback) circuits compared with the design involving only "combinatorial" (static) circuits. The accomplishment of such a system should be one of the principal objectives of the next PACT venture into the field of automatic programming.