Programming Techniques

M. D. McILROY, Editor

The ALCOR Illinois 7090/7094 Post Mortem Dump

R. BAYER
Boeing Scientific Research Laboratories, Seattle, Washington

D. GRIES
Stanford University, Stanford, California

M. PAUL
University of Illinois, Urbana, Illinois

H. R. WIEHLE
Telefunken, Konstanz, Germany

A dump technique for programs written in ALGOL 60 is described. This technique provides an intelligible analysis of an unsuccessful computation process in terms of the original source program.

Introduction

Much effort has been expended in recent years to provide powerful programming languages and sophisticated compilers in order to make communication between man and computer as easy as possible. This development, however, has completely bypassed one area where communication is particularly difficult, namely, post mortem dumps. Their purpose is to provide some information about what happened in the computation process if a program does not terminate successfully after it had started to execute. We will call such a situation from now on a *Failure*.

The techniques still widely used today in even the most modern and advanced third generation computers are obsolete and produce very inconvenient dumps. Such a dump consists typically of one cryptic message like "AC OVERFLOW AT LOC . . ." or "MEMORY PROTECT VIOLATION AT LOC . . .," and an octal or hexadecimal listing of certain storage areas containing the object code, library routines, and data. To analyze such a dump one must go through a cascade of decodings, know the main characteristics of a particular machine, its internal language, the operating (supervisory) system, and the object code corresponding to a program written in the source language, i.e., the mapping from the source language into the machine language performed by the compiler. This, however, is exactly the bulk of highly specialized knowledge that advanced programming languages and compilers are supposed to make superfluous. But even granted this knowledge, there still remains the difficult and tedious task of extracting from the mess of mainly superfluous information contained in a "dump" those clues which might contribute to a successful analysis of a Failure. It should be obvious by now that a better solution for the dump problem is most desirable.

Many solutions with various compromises between the completeness of the analysis and the cost in computer time and programming effort are conceivable; one solution which has been successfully implemented in connection with the ALCOR Illinois 7090 ALGOL-60 Translator is the subject of discussion in this paper. Since this dump routine has been designed for a specific language (ALGOL 60) and a particular class of machines (IBM 7090/94) and operating systems, some of the ideas described here have to be slightly modified for other configurations, but certainly the basic principles involved are almost universally applicable.

Objectives and Overall Organization

The following main goals had to be met in the design of the ALCOR Illinois 7090/94 Post Mortem Dump (PM-Dump):

1. Execution of an ALGOL program must not be slowed down if the program runs successfully.
2. Understanding the analysis given by PM-Dump must not require any specialized knowledge beyond what is necessary to write the ALGOL program.
3. The analysis should be dynamic as far as possible; i.e., events happening before the Failure with a possible influence on its cause should be recorded in the dump.
4. All information irrelevant to the determination of the cause of the Failure should be avoided.
5. Information referring to specific machine characteristics should be minimized.

PM-Dump is not a single piece of program, but splits naturally into several subproblems (see Figure 1) which are best dealt with at various stages (translation, loading, execution) of the complex process of running a program.

At translation time the compiler generates certain information, called Dump Information, which relates the

This project was supported in part by the Deutsche Forschungsgemeinschaft, in part by the Mathematisches Institut der Technischen Hochschule München, and in part by the Department of Computer Science of the University of Illinois

object code generated by the compiler to the original source program. The Dump Information is saved on some secondary storage medium like magnetic tape or disk and will be used at dump time by the routine PMDUMP.

It is useful, but not necessary, to provide a Loading Map containing, e.g., the name or the entry points, the total length, and the loading address (address of first instruction loaded in memory) of each routine in a job (consisting of the main program, code procedures, library routines, etc.).

At the very beginning of the execution of a job a short routine PREPM1 communicates with the operating system, indicating where control should be transferred in case of an unsuccessful termination.

After a Failure the operating system transfers control to a connecting routine PREM2 which saves the relevant content of the memory, i.e., mainly the object code and the working storage of the programs in a job. PREPM2 then calls the dump routine proper, PMDUMP.

PMDUMP uses the information saved by the preceding routines to determine the cause of the Failure, to give an analysis of what happened before it, and to print the names and the values of the variables defined at the time of the Failure.

It should be noticed that except for one call of PREPM1 the object code or the execution time of a program are not affected by PM-Dump. All preparations for a possible dump are made during compilation and an analysis of the computation process is carried out only after a Failure has occurred. In particular, no extra bookkeeping or tracing of the program control flow is involved at execution time.

Dump Information

The Dump Information is the crucial basis for the analysis of a Failure. It relates source program and object code. Since it is only available at translation time, it is saved by the compiler for PMDUMP. We will not describe here how a compiler should be organized to save the Dump Information. This problem is easy if kept in mind from the design stage on; it may be very difficult and tedious if an already existing compiler must be modified to generate the Dump Information. Here we will describe only what this information can typically consist of:

1. The PM-ID-list: a modified version of the ID-list, in which the entries for labels and name call formal parameters are deleted. (The ID-list contains, ordered according to program blocks, all identifiers in the program and necessary information about them, e.g., type, kind, number of subscripts, formal parameter or not, storage address [4].)

2. The PM-block-list: a version of the block-list modified to be compatible with the PM-ID-list. (The block-list contains in the ith location the triple (number of the block surrounding block i, pointer to that part of the ID-list corresponding to block i, number of identifiers in block i) [4].)

3. Localization-list (L-list): information about the object code, with one entry corresponding to each generated machine instruction. The L-list indicates which part of the source program the machine instruction corresponds to and what purpose it serves. In more detail this information consists of the following:

a. Number of the card in the source deck.
b. Block number in the ALGOL source program.
c. Field information, i.e., what type or part of a statement the instruction corresponds to:
   (i) procedure or function call,
   (ii) IF—THEN test,
   (iii) computation of an actual parameter,
   (iv) assignment statement,
   (v) array subscript calculation,
   (vi) array declaration,
   (vii) FOR statement.

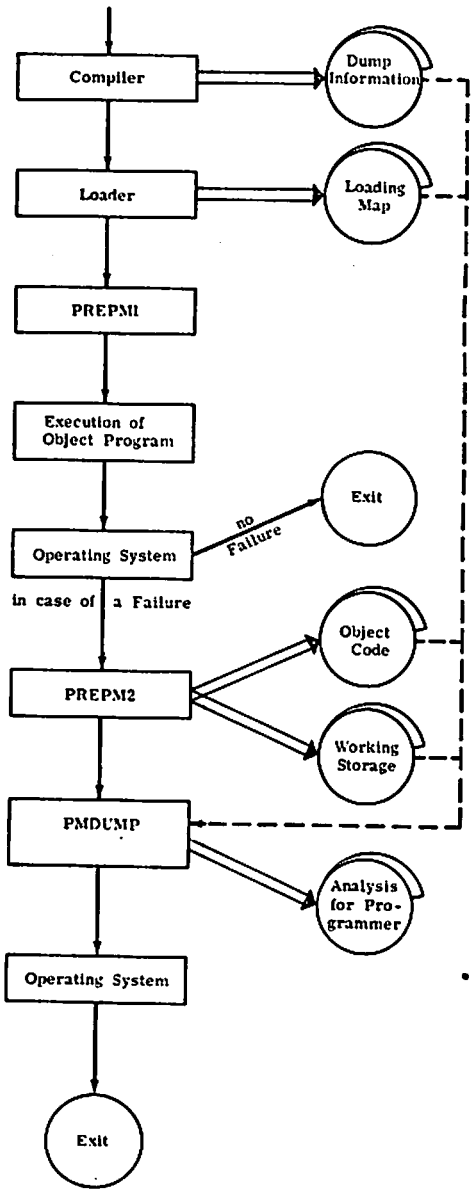

FIG. 1. Programs contributing to the ALCOR Illinois 7090/94 Post Mortem Dump

d. Operator information: the operator appearing last in the ALGOL source program.
e. Operand information: the type of operands (integer, Boolean, real) used with the last operator.
f. Failure point is or is not inside of a procedure or a special input-output procedure.

This information in the L-list about the Failure point will be printed out at dump time. Its purpose is to help the programmer pinpoint exactly how far the execution of the program had proceeded.

4. Information Record: It is convenient to generate this special piece of information containing a core load identification in the case of multiple core load jobs, the way in which the PM-ID-list, the PM-block-list and the L-list are stored in secondary memory, e.g., how many records or files exist on a magnetic tape. The main purpose of the Information Record is to allow efficient programming of PMDUMP. The details are not interesting for the purpose of a general description and are omitted here.

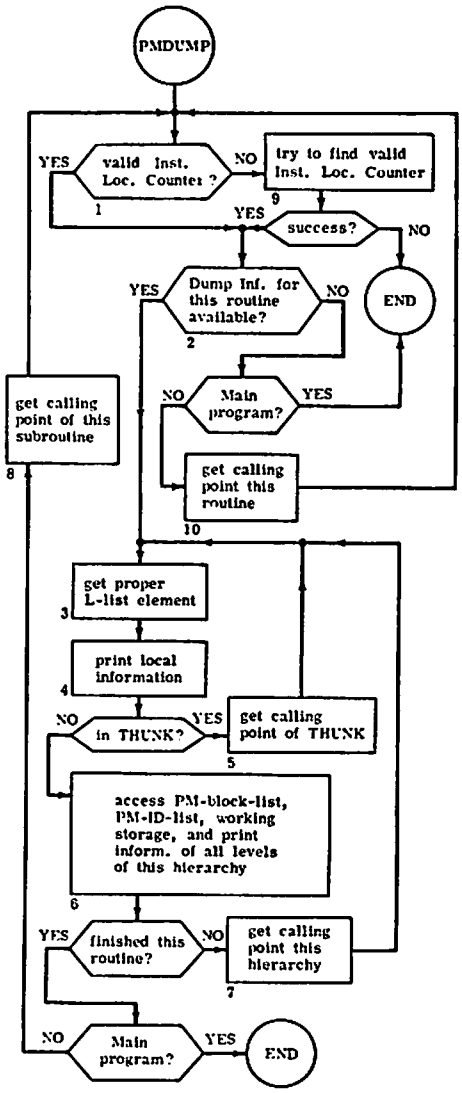

FIG. 2. Simplified flowchart of PMDUMP

Loading Map

The Loading Map is useful for finding out in which program (main program, code procedure, library subroutines) the Failure occurred, and how to get access to the proper Dump Information for this particular program. But there are other obvious solutions to this problem and we will not discuss it further.

PREPM1

This is the only routine which slows down execution if a dump is requested. But PREPM1 is a very short routine executing a few machine instructions (not more than 100), such as setting switches in the operating system, or initializing a few error returns. The loss of machine time due to PREPM1 is negligible.

PREPM2

Since PMDUMP is a relatively long program (approximately 3600 machine instructions), it should be loaded into the main memory of the computer only if needed. Thus a connecting routine, PREPM2, is used, to which the operating system transfers control in the case of a job which does not terminate successfully. PREPM2 saves the relevant content of the main memory, namely, the program and working storage areas for PMDUMP and passes certain information, e.g., the type of the Failure and the content of the Instruction Location Counter to PMDUMP. Then control is transferred to PMDUMP.

PMDUMP

A simplified flowchart of PMDUMP is given in Figure 2. Its main task is to use all the information made available to it by previous routines (see Figure 1), to coordinate it, and to present to the programmer as complete an analysis as possible of the state of the program and of its history at the time of the Failure. From this information the programmer should be able to derive very quickly and conveniently the reason why his program failed to execute properly.

Before we describe how PMDUMP accomplishes its analysis, one word of caution is in order: since the program being analyzed did not terminate successfully, something in the computation process must have gone wrong. Thus there is a possibility that the information passed on to PMDUMP is incorrect or misleading. This makes it necessary to check very carefully all the information coming from the main memory of the computer (object code and working storage) before using it. To give an example, suppose PMDUMP needs the address α of some transfer instruction of the form TRA α. Before using α a check must be made to determine whether the memory location in question really contains a transfer instruction and whether α is a legal address (e.g., does not refer to a protected area of memory).

PMDUMP provides the programmer with the following information:

(i) It prints local information about the Failure point

in the ALGOL program. This information is essentially contained in the L-list.

(ii) If the Failure point is not in hierarchy 0 (the hierarchy number is defined as the level of nesting of procedures starting with 0 for the main program [4]), PMDUMP prints the name of the procedure in which it is contained.

(iii) Starting with the innermost block containing the Failure point and proceeding to surrounding blocks, both the names and the present values[1] of the variables defined in these blocks are printed out. This process is continued until all the blocks in some given hierarchy have been dealt with. If the hierarchy is 0, PMDUMP terminates; otherwise it determines the point from which the procedure was called; this calling point is then considered as the new Failure point, and steps (i), (ii), and (iii) are reiterated.

We now describe in more detail the blocks of the flowchart in Figure 2 and how steps (i), (ii), and (iii) are accomplished.

(1) It is assumed that PREPM2 or the operating system delivers to PMDUMP the Instruction Location Counter indicating where the computation was terminated. One must check whether the Instruction Location Counter is valid (e.g., points to a location in the object code area of the memory).

(2) Using the Loading Map the routine in which the breakdown occurred, say R, is determined.

(3) If the compiler has generated any Dump Information for R we use the 1–1 correspondence between the object code of R and the L-list for R to find the entry of the L-list corresponding to the Failure point, and

(4) Decode and print out the local information contained in it.

(5) A THUNK is a piece of code which is used to compute an actual parameter [2]. If the Failure point is in a THUNK, then we must find the calling point of the THUNK, i.e., the point where the actual parameter being computed by the THUNK is used. This calling point lies always deeper in the nested structure of an ALGOL program than the THUNK itself. Notice that in this case we have to reiterate step (i) instead of proceeding to step (iii). Finding the calling point of a THUNK requires a tricky and detailed analysis of the object code corresponding to it. Since this problem is highly dependent on ALGOL and a specific implementation of it by a compiler, we will not pursue the matter any further except to mention that the difficulty in THUNK analysis is the following: Generally in the analysis of an ALGOL program we proceed from the inner parts of the nested structure to the outer parts; if we analyze a THUNK, however, we have to reverse this direction momentarily to get deeper into the program and then to come back out again.

(6) The block number found in the L-list entry in (3) and (4) and the PM-block-list determine that part of the PM-ID-list which contains the names and addresses of the variables declared in the present block. We use the address to retrieve the present value of the variable from the working storage saved by PREPM2 and print it out together with the name. After the block is exhausted and if the surrounding block is in the same hierarchy, we reiterate (6) for the surrounding block.

(7) If the surrounding block is in a lower hierarchy than the present block, then we must be inside of a procedure. Each procedure has associated with each (possibly recursive) call a relocatable working storage (called FFS in [2] and [4]) which is dynamically allocated at run time. The relocation amount of the FFS is called the Stack Reference [4]. The FFS contains the calling point of the procedure and the Stack Reference of the calling hierarchy [2]. We retrieve both of them; the calling point becomes the new Instruction Location Counter, and the Stack Reference is used to access the FFS of the calling hierarchy in the next reiteration of (3). For reasons mentioned earlier again a detailed analysis (as in (9)) of the FFS of both hierarchies is necessary to establish confidence in the retrieved information.

(8) If the analysis of the present routine is completed and this routine was the MAIN program (which can be found out, e.g., from the Loading Map or by looking at the object code), then PMDUMP has accomplished its task. Otherwise we have been analyzing some subroutine. We then find the calling point and the corresponding Stack Reference similarly as in (7) and reiterate PMDUMP itself.

(9) If we do not find a valid Instruction Location Counter in (1), we may still try to continue the analysis. One possibility is to use the Stack Reference delivered to PMDUMP and to interpret the corresponding FFS, which should have a well-determined structure, e.g., it should contain a Stack Reference, a calling point, and certain standard sequences of instructions. If this is not the case, then only the outermost block of the program can be analyzed; otherwise it is quite plausible that we have found an FFS corresponding to a procedure call in some historic moment of the program, and although we have no local information, we can still attempt to give a partial analysis of this procedure. Then we certainly can find the calling point of the procedure and continue the analysis from there.

(10) It can happen, of course, that a breakdown occurs in a routine for which no Dump Information is available, e.g., for a library subroutine or a procedure coded in some language other than ALGOL. The possibilities of analyzing such a case depend mainly on how the rest of the operating system is designed. It is optional to skip the dump of that routine entirely and to continue the analysis from the calling point of the routine, or to give an octal dump of the routine and its associated working storage.

[1] At a slight cost in execution time the working storage can be initialized with an "illegal" number, e.g., an unnormalized floating-point number. Variables which have not yet been assigned a value at the time of the Failure can then be printed out as "undefined" (see Figure 3).

```
$      ALGOL, GO, PMDUMP, CLEANUP


                    ALCOR- ILLINOIS 7090 ALGOL COMPILER

1       'BEGIN'  'COMMENT'  EXAMPLE OF AN ANALYSIS BY THE ALCOR ILLINOIS        JOB ONE
2            7090/94 POST MORTEM DUMP .,
3            'REAL' A.,  'INTEGER' B., 'BOOLEAN' C., 'ARRAY' MATRIX(/1..7/).,
4            'REAL''PROCEDURE' REALP(X,Y)., 'REAL' X., 'INTEGER''PROCEDURE' Y.,
5                 REALP .=X*3+Y(X).,
6            'INTEGER' 'PROCEDURE' EXTERN (X).,  'REAL' X.,  'CODE'.,
7            MATRIX(/3/) .=3.7.,  A .=1000.,  C .='TRUE'.,
8            'IF' A 'GREATER' REALP(A*A,EXTERN) 'THEN' B .=REALP(A*A,EXTERN).,
9            'END' 'FINIS'


$ ALGOL


                    ALCOR- ILLINOIS 7090 ALGOL COMPILER

1       'INTEGER' 'PROCEDURE' EXTERN(X).,  'REAL' X.,                          JOB ONE
2       'BEGIN' 'INTEGER' I.,  'REAL' R.,  R .=5.,
3            'FOR' I .=1 'STEP' 1 'UNTIL' R*3*X 'DO' R .=R*X.,  EXTERN .=R.,
4       'END' 'FINIS'


                    ALCOR-ILLINOIS-7090 POST MORTEM DUMP

PM-DUMP CALLED BECAUSE OF FLOATING POINT TRAP
HANDUP OCCURRED IN ROUTINE    EXTERN,CORE LOAD JOB ONE
     INSIDE OF PROCEDURE     EXTERN
CARD NUMBER IS     3      IN BLOCK NUMBER    3
PROGRAM WAS INSIDE OF
     A FOR STATEMENT
THE LAST EXECUTED OPERATOR WAS ..  *
THE FIRST OPERAND WAS OF TYPE 'REAL'.     THE SECOND OPERAND WAS OF TYPE 'REAL'.
DUMP BLOCK NO.     3
     I      =     7                  R        =     .49999999E 37
DUMP BLOCK NO.     2
     EXTERN  =     UNDEFINED
CALL CAME FROM    (MAIN)
     INSIDE OF PROCEDURE     REALP
CARD NUMBER IS     5      IN BLOCK NUMBER    2
PROGRAM WAS INSIDE OF
     A PROCEDURE OR FUNCTION CALL
     AN ASSIGNMENT STATEMENT
THE LAST EXECUTED OPERATOR WAS ..  *
THE FIRST OPERAND WAS OF TYPE 'REAL'.     THE SECOND OPERAND WAS OF TYPE 'INTEGER'.
DUMP BLOCK NO.     2
     REALP   =     UNDEFINED
CALL CAME FROM    (MAIN)
CARD NUMBER IS     8      IN BLOCK NUMBER    1
PROGRAM WAS INSIDE OF
     A PROCEDURE OR FUNCTION CALL
     AN IF - THEN TEST
THE LAST EXECUTED OPERATOR WAS ..  *
THE FIRST OPERAND WAS OF TYPE 'REAL'.     THE SECOND OPERAND WAS OF TYPE 'REAL'.
DUMP BLOCK NO.     1
     A       =     .09999999E 04     B        =     UNDEFINED        C        =     'TRUE'
IN ARRAY PRINTOUTS SUBSCRIPTS BEGIN AT THEIR LOWER BOUNDS. THE N'TH SUBSCRIPT RUNS THROUGH ALL PERMISSIBLE
VALUES BEFORE IT IS RESET TO ITS LOWER BOUND AND THE N-1'TH SUBSCRIPT INCREMENTED.
     ARRAY        MATRIX
      UNDEFINED                UNDEFINED                .370CC000E 01            UNDEFINED                UNDEFINED
      UNDEFINED                UNDEFINED
```

FIG. 3. Post mortem analysis of an ALGOL program

Conclusion

Figure 3 is a typical example of an unsuccessfully executing ALGOL program analyzed by the ALCOR Illinois 7090/94 Post Mortem Dump. The transparency of the analysis should make any further explanation redundant. We leave it to the reader to compare it with octal or even hexadecimal listings of the contents of computer memories and to draw his own conclusions.

Acknowledgment. The authors wish to thank J. Cohen and R. Penka from the University of Illinois for their most valuable contributions in the planning and programming of the ALCOR Illinois 7090/94 Post Mortem Dump, which has proven that the implementation of the ideas presented in this paper is feasible, economical, and useful.

RECEIVED MARCH 1967; REVISED AUGUST 1967

REFERENCES

1. BACKUS, J. W., ET AL. Revised report on the algorithmic language ALGOL-60. *Comm. ACM 6* (Jan. 1963), 1–17.
2. BAYER, R., MURPHREE, E., JR., AND GRIES, D. User's manual for the ALCOR ILLINOIS 7090 ALGOL-60 Translator. 2nd Ed., Dept. of Comput. Sci., U. of Illinois, Urbana, Ill., Sept. 1964.
3. BAYER, R., COHEN, J., AND PENKA, R. The ALCOR Illinois 7090 Post Mortem Dump, description and imbedding instructions. Rep. No. 198, Dept. of Comput. Sci., U. of Illinois, Urbana, Ill., Feb. 1966.
4. GRIES, D., PAUL, M., AND WIEHLE, H. R. Some techniques used in the ALCOR Illinois 7090. *Comm. ACM 8* (Aug. 1965), 496–500.
5. ——, ——, AND ——. ALCOR Illinois 7090. An ALGOL compiler for the IBM 7090. Rep. No. 6415, Rechenzentrum der Technischen Hochschule München, Germany, 1964.