The criterion COBOL system

by MICHAEL D. SHAPIRO
NCR Corporation
San Diego, California

INTRODUCTION

Higher level programming languages provide problem solvers an access to computers without requiring them to become computer experts. In the traditional environment, compilers perform the relatively complex translation from the higher level language statements into the machine instructions. The differences between the complex data structures and operations of languages and typical simple structures of machines makes this a wide chasm to bridge.

The mismatch between data and operations of the languages and machines appears clearly when one considers a language such as Cobol, APL, PL/I or Snobol4. In each of these, the fundamental operations and even basic data types do not translate easily because the language design depends on no one machine. The implementor must build mechanisms ranging from full software interpreters to elaborate subroutine libraries to perform the higher level language operations on higher level language data.

Despite the costs, the payoffs of using higher level languages make them worthwhile. And with the widespread use of microprogrammable processors as the base of computing systems, designers can realize the once primarily theoretical higher level language machines. An interpreter for the operations of a higher level language, written in firmware rather than software becomes a higher level language machine.

The NCR Criterion Cobol System implements an indirect higher level language machine for 1974 ANSI Cobol in a conventional microprogrammed environment, permitting execution of a single run unit composed of both Cobol and other language modules in a highly efficient manner.

The system consists of a microprogrammed hardware processor supporting both conventional and Cobol machine firmware in a multiple virtual machine environment, a multiprogramming virtual resources operating system, a 1974 ANSI Cobol compiler designed to run in a virtual storage environment, the necessary execution time support subsystems and routines, and Cobol symbolic debug and runtime tuning subsystems.

The discussion focuses on the characteristics of the system as a Cobol environment and the structure of the Cobol virtual machine, the compiler, and the support subsystems.

THE CRITERION SYSTEM

NCR system architects chose a higher level language approach as part of the basic design of the Criterion, a new family of computers to replace the aging Century line. The other part of the design fully supports, without modification, existing NCR Century application programs, no matter what their programming language.

Cobol, probably the most widely used data processing applications language and hence of most interest to the company, serves as the base of the higher level language system. The new virtual resources executive (VRX) operating system design includes support of multiple virtual machines. Cobol Virtual Machine (CVM) firmware, a new American National Standard Cobol compiler, and a set of Cobol execution time support subsystems form the resulting Criterion Cobol System (CCS).

A very fast (56 nsec) pipelined microprogrammable processing element forms the base hardware of the Criterion system. Three storage units attach to it: register storage (RSU), instruction storage (ISU) and memory storage (MSU). The RSU holds temporary work areas for the element. Microcode executes primarily out of the ISU, although some low-speed instructions execute from the MSU. The MSU areas not used by firmware become the main storage used by the operating system and application programs.

The hardware supports a paged memory environment with a maximum addressing capability of 16 megabytes (MB). The processing element performs address translations with interrupts to the executive on page faults.

The customer can receive either of two sets of firmware and software for the Criterion: the real resources system (RSI) or the virtual resources system (VSI). RSI emulates NCR Century computers and permits use of the Criterion as a direct replacement. VSI supports the virtual resources, including virtual storage and the multiple virtual machines. The two current virtual machines provide an extended version of the Century instruction set and architecture, supporting the NEATVS assembly language (NVM) and the new CVM.

While these two machines co-reside in the system and support many similar elementary data types, they have sev-
eral fundamental differences in design orientation. NVM uses binary data extensively while CVM works best with decimal data. NVM uses an absolute addressing scheme, but with addressing a maximum of 32 kilobytes (KB) offset from a register, except in the first 64 KB. CVM uses bases, but allows offsets up the full virtual address space limit.

Both NVM and CVM share a common stack-oriented calling mechanism and a common virtual addressing scheme so that a program executing in either machine mode can invoke a module in the other. Each entry point to a module has an entry point control item (EPCI) indicating the virtual machine mode for execution of that module. Machine state can change on any call.

COBOL VIRTUAL MACHINE

The Cobol Virtual Machine (CVM) implements an indirect higher level language machine for the Cobol data types and structures and for a large subset of the Cobol operations. An object module generated for execution by the CVM contains three principal memory areas and some minor auxiliaries.

The data area contains all the data described in the program either explicitly or implicitly. Implicit data includes various file tables, collating sequence tables, debugging tables, and literals. Data definition entries in the Data Division describe explicit data.

The descriptor area contains an internal representation of the information contained in the data description entries. Each referenced data element has a descriptor. Each descriptor contains location, size, type, index, and editing information for the data element.

The code area contains the translated procedure division statements. The basic internal operations (for example, simple MOVE or ADD) translate to single machine instructions. More complicated statements, such as those having multiple receiving fields, decompose into simple Cobol statements for code generation. With operation codes generally typeless, automatic conversion and editing occur when required by the referenced descriptors.

Data types

The fundamental elementary data types in the CVM correspond to the variations on COMPUTATIONAL and DISPLAY types in Cobol.

Packed decimal items represent computational data internally. The requirement for frequent conversions makes packed decimal the most efficient format to use in CVM. Depending on the Cobol picture, the item may or may not have a sign. Computational data items may vary in precision from 1 to 18 digits and may have an implied decimal point at any position adjacent to a digit or may have scaling to a maximum of 18 positions from the farthest digit. An additional four-bit trailing half-byte represents the sign, when present.

Both numeric and nonnumeric data items occur. Numeric display items follow the same rules for computational items on precision and decimal point position. ASCII characters ‘0’ through ‘9’ represent the digits. If the programmer specifies a signed item with a “separate sign,” an ASCII ‘+’ or ‘−’ represents the sign. The user specifies whether the sign goes before or after the digits. If the user requests a “zone sign,” the sign indicator combines with the first or last digit to form an ASCII character according to this table:

<table>
<thead>
<tr>
<th>No sign</th>
<th>+ sign</th>
<th>− sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>A B C D E F G H I</td>
<td>J K L M N O P Q R</td>
</tr>
</tbody>
</table>

Strings of ASCII characters form the nonnumeric data items, having the internal categories of alphanumeric, alphanumeric justified right, numeric edited, and alphanumeric edited, and the properties specified in the Cobol standard.

The CVM provides two additional data types. Signed binary (two, four, or eight bytes) provides data communications with the NVM. User documentation recommends against its use for other purposes. Special index-handling commands manipulate index data items for the table handling operations. The internal structure of these items remains transparent to the user.

Descriptors

The descriptor area may contain any combinations of four types of descriptors. Two of the descriptors, used for compact data descriptions, require four bytes each. Only certain elementary items use them. The other two descriptor types define the more complicated data structures. The long descriptors take eight bytes each.

Short nonnumeric descriptors describe alphanumeric or alphanumeric justified right data items whose length does not exceed 4095 characters. Long numeric descriptors describe decimal, packed decimal, or binary items. In both cases, the item must begin in the first 32768 bytes of the data area.

Long nonnumeric descriptors describe alphanumeric, alphanumeric justified right, or alphanumeric edited data items. They allow a maximum length of 65535 characters. Other fields describe the number of dimensions the item has, whether an edit information mask pointer follows, and the method of addressing used for the data. Long numeric descriptors vary from the long nonnumeric descriptors in that they have decimal position and length fields in place of character length fields.

Edited data items require edit information. An edit mask provides the description of the format of the result. The mask derives from the picture specified in the source program. For alphanumeric edited and numeric edited items, a flag in the descriptor indicates an extension to twelve bytes, with the additional four bytes pointing to the edit mask and containing the numeric length for numeric edited items.

Addressing

Addressing in the CVM uses the standard 24-bit Criterion virtual address mode. Within a program, the system com-
COMMENTS are part of the instructions, where appropriate: numeric operations have rounded and size-error flags.

A few of the typical commands demonstrate the nature of the CVM instruction set:

**MOVE** commands act like simple move statements in Cobol, reformatting data as necessary from the source to the destination. A single instruction provides for all conversions and editing. The MOVE-BYTE commands allow movement of a single character literal.

**COMPARE** commands form the basic operations of the IF statements. The principal COMPARE command handles both numeric and nonnumeric data. Separate instructions provide IF-ALPHABETIC and IF-NUMERIC tests. Comparison tests use either native ASCII collating sequence or one derived from some table, such as EBCDIC. The user specifies the choice with a PROGRAM COLLATING SEQUENCE clause.

**GOTO** commands provide for internal branching, with Cobol GO TO and GO TO... DEPENDING statements as basic instructions. Others include a conditional GO TO for size error detection and an indirect GO TO for PERFORM handling.

Arithmetic commands mimic the Cobol arithmetic statements (except the COMPUTE statement). The commands require numeric source operands and numeric or numeric edited result items. Both two- and three-address operations occur, corresponding to statements of both forms:

```
ADD A TO B.
ADD A, B GIVING C.
```

Programmers request rounding and size error detection in the arithmetic commands by clauses in source:

```
ADD A, B GIVING C ROUNDED;
ON SIZE ERROR...
```

**CALL** commands invoke a standard Criterion calling sequence. The program sets up parameters in the standard parameter stack using a MOVE-VMA command to compute the virtual address. A state switch to NVM may occur during the call.

**EXIT** commands provide the Cobol EXIT PROGRAM statement for a called subprogram. On exit to the calling program, a state switch may occur to return to the NVM state if an NVM program invoked the CVM program.

**THE COBOL COMPILER**

The Cobol compiler (CBL) for the CCS converts 1974 ANSI Cobol programs to CVM object modules. Its logic design generally matches that of a typical textbook example Cobol compiler, but with the unique feature of its extensive use of the Criterion virtual address space.

The implementation team used contemporary software engineering techniques in constructing the compiler. They wrote the bulk in a subset of the NCR Software Writer's Language (SWL). NEATVS assembly language functions provide a few utility services, and Cobol coded modules (after bootstrapping) do the input-output. CBL contains more than 200 modules, linked together in a tree-structured manner.

The development process, which occurred concurrently with the development of the Criterion and the operating system, heavily used a variety of tools. A commercial timesharing system served as the initial development engine. An earlier Cobol development project formed a base for about 30 percent of the code. Snobol4 encoded programs translated this code to the SWL syntax required by CBL. The compiler team coded the remainder of the compiler in the SWL subset. A tools group developed a compiler for the time-sharing system to match the subset of SWL used by CBL. The resulting object program for CBL executed under a simulated virtual storage test environment developed for the CBL project. The team debugged most of the compiler logic before the new hardware and operating system became available.

When the Criterion system could run with a rudimentary operating system, a modified remote entry system transmitted source modules to the NCR equipment for compilation and testing. This linkage remained in place throughout the development cycle. The master library remained on the time-sharing system until the Criterion system became fully stable.

**User's view of CBL**

To the Cobol programmer, CBL appears as an easy-to-use conventional compiler. Information provided at compilation time relates the execution time debugging output to the source program. The compiler can produce extensive diagnostics in clear English. The programmer specifies a
requested minimum level, from non-ANSI feature through serious errors only. Diagnostics refer to the source program by source line and column number.

Input to the compiler comes from source cards, from a Cobol source library, or (for compatibility with earlier systems) from NCR "SPUR" format files. Outputs include the source listing and a diagnostic summary, along with the object module on disc, if the compiler generates it. The user may specify optional cross-reference, object listing, and object map printed output. The Cobol library facility furnishes texts for the Cobol COPY statement.

Object modules produced by the compiler contain the code, data, and descriptor tables required by the CVM, the control tables used by the link editor, and optionally, debugging tables used by the execution time Cobol symbolic debugging and tuning systems.

**Tables and texts**

The structure of internal tables and texts gives CBL the properties which make it particularly well-suited for virtual storage operation. As used in this discussion, a table structure holds randomly accessed data and a text structure stores sequentially accessed data.

Tables in CBL contain the information usually associated with the compilation process: data names and attributes, procedure (label) names and attributes, file properties, literals, etc. In keeping with the rules for virtual storage programming, the design of tables makes them compact. A vector of data records forms a table. If a record requires eight or fewer bytes, the table consists of the collection of records. If the record size exceeds eight bytes, the table contains a vector of three-byte record pointers and a collection of records allocated, as required, from a storage pool. This approach permits CBL to compile the widest mix of Cobol programs, keeping the table storage as compact as possible.

A few principal internal tables hold most of the data passed between phases:

- **DATA TABLE.** The data table contains the data name information and the attributes of the data name. The attributes include both external properties, such as line and column where defined, and internal properties, such as data type or level in structure. An auxiliary table entry holds less frequently used items which cannot fit into the data name attribute table.

- **PROCEDURE TABLE.** The procedure table contains the procedure name (Cobol paragraph and section label) information. It uses the auxiliary table when necessary.

- **PICTURE AND LITERAL TABLE.** The picture and literal table contains pictures and literals which appear in the source program, literal generated by the compiler, and edit masks generated from the pictures for numeric edited and alphanumeric edited data items.

- **FILE TABLE.** The file table takes the information developed by the compiler from environment, data, and procedure division references to files. The file table links to the data table and to the auxiliary table to store additional information.

Texts in CBL store the information frequently associated with temporary scratch files. A text appears as a large vector of relatively small records. The compiler places entries into a text or retrieves entries from a text in a sequential manner. In this way the virtual storage paging mechanism moves the idle information to the page storage device and brings it back when referenced. The overhead associated with normal file input-output operations disappears and, for a small Cobol program in a relatively idle system, the text pages may never need paging out.

The principal internal texts used for passing information between phases in CBL are:

- **COBOL TEXT.** The Cobol text (CTEXT) contains a regularized representation of the original source program with data names, procedure names, and reserved words, pictures, and literals represented by entry numbers in appropriate tables.

- **GENERATOR TEXT.** The generator text (GTEXT) contains an intermediate form of the Cobol program, with references to table entries for data items, labels, literals and other items. The GTEXT contains a series of "packets" which correspond roughly to assembly language statements in a compiler which compiles to assembly language in the "front end."

- **OBJECT TEXT.** The object text (OTEXT) contains the object information produced by the object generator, but before packaging into an external object module format.

- **ERROR TEXT.** The error text (ETEXT) contains the information produced by the compiler analysis phases and used by the diagnostic formatter.

**Compiler phases**

CBL divides into from five to ten phases, depending on how one wishes to number them. If one chooses the minimum number, several of the phases divide into subphases.

A main driver initializes compiler work areas, opens source and listing files, processes user request options and then calls the first compiler phase. It proceeds through each compilation phase, checking for a possible compilation abandonment after each, until it reaches the object generation phase. It invokes generation only if:

a. the user has not suppressed it (an option), and
b. no serious error has suppressed it.

After object generation, the object formatter outputs the code, the main driver prints a compilation summary, closes
The source scanner, first phase of the compiler, converts the Cobol source lines read from input to the CTEXT. A hashing mechanism (buckets and chained entries) builds the symbol table through the data and procedure tables. Pictures, literals, and unknown symbols go into the picture and literal table. The scanner produces the source listing and places syntax diagnostics into the ETEXT.

The source scanner does all the Identification Division processing, principally recording the program name.

Three subphases handle the Environment and Data Division processing. One of these, the picture analyzer, scans the picture strings for meaning and errors, and sets up attributes for data associated with the picture. It also builds edit masks corresponding to specifications in the picture. Two subphases extract source from the CTEXT and parse and analyze other table information in the divisions, building and modifying the appropriate table entries. All three subphases can generate diagnostic requests into the ETEXT.

The Procedure Division processor takes the source text from the CTEXT and information from the tables and builds the GTEXT. It also makes entries, if necessary, in the ETEXT and other tables.

The final front-end phase, the literal pooler, moves entries from the picture and literal table into the literal pool, a small text.

Two back-end subphases generate the object module. An object generator analyzes the tables and literal pool to build the data and descriptor areas. It uses the results from these and the GTEXT to construct the code area, and also produces the required link editor tables. All of this information goes into the OTEXT, which passes to the object formatter. The object formatter writes the object module on disc and optionally produces an object map and listing.

The main program has two utility phases available when it needs them. A cross-referencer produces an alphabetic reference listing of all data and procedure names. The main program runs this procedure, on user request, immediately after the source scanner. The diagnostic formatter runs just before the end of compilation if any phase has emitted a diagnostic request to the ETEXT.

EXECUTION TIME SUPPORT

Criterion Cobol System execution time support includes a set of "global" routines for handling the operations required for input-output, sort-merge, communications, and operating system interface statements. It also contains software/firmware debugging and tuning subsystems, described in the next section.

Global routines in VRX, residing in a protected memory area accessible to all simultaneously executing programs, consist of reentrant code.

The input-output system, the Criterion Access Method (CAM), provides Cobol support for sequential, relative, and indexed files. Relative and indexed files may reside on any disc type supported on the machine. Other devices handle sequential files only.

The sort-merge system provides the Cobol support for sorting and merging magnetic tape and disc files.

The Message Control System (MCS) provides functions for the Cobol Communications functional module. The Network Description Language (NDL) processor sets up the communications paths according to user specifications.

An operating system interface module, invoked primarily for Cobol ACCEPT and DISPLAY statements, supports such actions as reading the clock or calendar, communicating with the operator, and reading control card parameter information.

COBOL SYMBOLIC DEBUG AND TUNING AIDS

The Cobol symbolic debug system (COBUG) provides the Cobol programmer with a variety of information for symbolically monitoring the program and data flow in an executing program.

The first instruction of each paragraph contains a special flag bit. Before executing the instruction, the firmware checks that bit and an internal symbolic debug switch. With both conditions on, the COBUG subsystem takes control. It displays information requested by the user. The user, through control statements, may request a variety of types of information:

DUMP of all or selective data items at specific places in the program. Options include specification of which time during execution the dump should occur, the number of times, and relational conditions:

DUMP ABC AT XYZ FOR 10 TIMES WHEN A IS LESS THAN B

TRACE of all or selective paragraph flow. Options include ranges, and the number of times:

TRACE XYZ TO PDQ ON 5 FOR 12 TIMES

FLOW of program execution. If requested, the firmware records the execution of each instruction. At program termination time, the COBUG system formats and prints a listing of the data. The user, who must understand the CVM operation, can use this listing to detect problem points in the program.

The Cobol analysis routine for runtime tuning (CARTUNE) provides the Cobol programmer with a mechanism for collecting execution time statistics. The Cobol compiler, on request, builds a source line table and three activity tables in the object module. At execution time, the CVM firmware optionally maintains tables of the number of times each type of operation code executes, the number of references to each data item, and the number of times each instruction executes.

At the end of execution, CARTUNE gathers the information and prints an analysis. The user may request specific
reports: source line executions, data references, and object instruction execution history. These data assist the programmer in locating the inefficient portions of the program for improvement.

THE RESULT

The Criterion Cobol System provides the Cobol user with an integrated system optimized for highly efficient Cobol program development and execution.

ACKNOWLEDGMENTS

CVM designers and implementers:
  Ward F. Hardman, Jr.
  C. Kent Huckstep
  Donna G. Hynes
  Donald G. McCrimmon
  John D. Roberts
  William F. Schweitzer
  Michael D. Shapiro
  Robert A. Surtees
  Wayne H. Uejio

Cobol compiler team:
  William A. Aebi
  T-C Chao
  Charles L. Citron
  Dana M. Foy
  Donna G. Hynes
  Donald G. McCrimmon
  Neal F. Openshaw
  Ray L. Paul
  Don A. Schricker
  Michael D. Shapiro
  Dale A. Shoemaker
  Robert A. Surtees

Cobol symbolic debug and tuning aids:
  Dana M. Foy
  Larry R. Luko

REFERENCES