Partial recompilation*

by RONALD B. AYRES and RICHARD L. DERRENBACHER**

Technology for Information Management, Inc.
Albany, New York

INTRODUCTION

It is axiomatic that a great deal of the machine time required to debug and implement a computer program is used to compile and recompile source code. In many cases, recomplilations require at least as much machine time as an initial compilation, although the resultant change to the program object code may be minimal. In a time-sharing environment the redundancy associated with these recomplilations can be expected to contribute to deterioration of response time. Conversational debugging packages, such as DDT,1 seem to present only a partial solution to the problem, since the debugging operation is usually divorced from the source language updating and production of a corrected object code program.

The purpose of this paper is to describe a compiler writing technique which eliminates the need for entire program recomplilations. The technique is known as "partial recompilation" and has been implemented into the JOVIAL† compiler for the 1604B computer at the U.S. Air Force's Rome Air Development Center (RADC). While interpreters and incremental compilers2,3,4 have been designed primarily for time-sharing systems, the technique described in this paper may be used in a batch environment as well, particularly when used to debug and implement very large programs which require large amounts of compilation time. The subject compiler was modified to incorporate conversational features and to operate in either a batch or in a simulated time-shared environment.

APPROACH

In the classic case, computer programs are written, debugged, and implemented as follows:

1. Write program source code statements.
2. Compile source program and produce object code.
3. If errors are detected, correct source code statements and return to step 2.
4. Execute program object code.
5. If errors are detected, prepare corrected source code statements and return to step 2.

Programmers who are fortunate enough to know machine code often eliminate the requirement to return to step 2 by modifying the program object code and continuing with step 4. Although this short cut often saves time, several pitfalls confront the programmer who elects to debug in this manner. First of all, errors are probably more apt to be made when program corrections are introduced at the machine code level. Secondly, the possibility always exists that the programmer will forget to implement a correction at the source level after it has been successfully implemented and tested at the object code level. Finally, as source language corrections are accumulated over long time spans, the possibility of the introduction of new errors is magnified.

Conversational debugging packages† which permit source level corrections without necessitating a program recompilation provide a similar debugging shortcut and eliminate the need to enter corrections at the machine code level. Even with such debugging packages, the other pitfalls still exist, however. Certain types of program modifications are not possible at the object code level at all, e.g., expanding or reducing the size of certain tables or buffer areas.

The technique of partial recompilation reduces program debugging time by permitting any source

---

* This work was performed at TIM, Inc. and was supported entirely by the U.S. Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York under contract F30602-69-C-0316.

** Currently associated with Mohawk Data Sciences and Informatics Incorporated, respectively.

† JOVIAL is a general purpose, relatively machine independent, procedure oriented language developed by System Development Corporation. It is the official command and control language for the U.S. Air Force.
language modification to be made while nearly eliminating the requirement to completely recompile an entire program, i.e., by eliminating the requirement to ever return to step 2 of the program implementation process. Instead, programs are implemented as follows:

1. Write program source code statements.
2. Compile source program and produce object code.
3. If errors are detected, correct source code statements and compile only those statements necessary to effect the change in the previously generated object code.
4. Execute program object code.
5. If errors are detected, correct source code statements and compile only those statements necessary to effect the change in the previously generated object code and return to step 4.

Thus, a means of performing a partial recompilation is provided. Since the partial recompilation results in the processing of only a small number of source language statements (depending upon the number and type of source language modifications made), a significant amount of time is saved over performing an entire program recompilation. For most types of source statement modifications, the partial recompilation takes only a fraction of the time required to perform an entire program recompilation, since most of the redundancy associated with program recompilations is eliminated. The programmer need not learn machine or object code, since all communication with the compiler takes place at the source language level.

COMPILER OPERATION

Source language input records contain an alphanumeric source record identification, or “sequence number.” This record identification is unique and is used to refer to a unique program record, or line. In addition to the usual editing and modification functions, the sequence number is used by the compiler as a link between the source language input statements and the intermediate information generated and saved for those compiled statements. This relationship is necessary in order that the compiler may determine exactly which source language statements must be compiled (or recompiled) based upon modifications made to a source language program.

The partial compiler operates in one of two modes, as specified by control information provided by the user. The first mode is designated the INITIAL mode and operates very much like a classic compiler. An INITIAL compilation is always performed when a program is compiled for the very first time. During this compilation, all source language statements comprising the program are compiled. The second mode is the PARTIAL mode and is used whenever source language modifications are to be made to a previously compiled program.

The INITIAL compilation is performed as follows:

1. Generator Phase

Source language statements are obtained from either a prestored source language file or are transmitted directly to the compiler via an on-line CRT unit. If the statements are transmitted via the CRT unit, a source language file is constructed. As the statements are obtained, the statement sequence numbers are used to construct entries in a Sequence Number Dictionary Table (SNODCT table). Source language is processed on a statement by statement basis. If a syntactic error is detected and the compilation is being performed in the time-sharing, rather than batch environment, the user is notified via the CRT unit and is given the option of correcting the error immediately, proceeding, aborting the compilation, or obtaining “tutorial” assistance in identifying the detected error. Statements are translated into binary values which contribute to the construction of various compiler lists and tables, such as the dictionary and intermediate analysis tables. The SNODCT table is constructed so that the intermediate data generated for a given source language statement may be identified.

2. Translator Phase

Declarative data from the dictionary is translated into machine language via entries in an intermediate object code table. This consists of reserving core storage, presetting the environment of the object program, processing long octal and literal constants, and modifying the dictionary. Secondly, imperative data from the intermediate analysis tables is translated into machine language via entries in the intermediate object code table. Temporary registers are allocated and tag locations entered into the dictionary.

3. Assembly Phase

The assembly phase generates an object code program from entries in the intermediate object code table. The SNODCT table is completed so that the object code program registers generated for a given source language statement may be identified.
The most noticeable difference between the classic compiler and the INITIAL mode of the partial compiler is that the partial compiler saves the generated data, such as the dictionary and intermediate analysis tables, and generates the SNODCT table as a means of relating source language statements to the compiler generated data and to the object code program registers generated for those statements. All such output from the INITIAL compilation is written on the secondary storage device and is required as input for a PARTIAL recompilation. The overhead associated with the generation of this data during an INITIAL compilation is usually more than compensated for by the time saved during the first PARTIAL recompilation. This overhead was found to never exceed 15 percent for the 5 JOVIAL programs for which timings were made. The larger mass storage requirements are imposed upon the system by the partial recompilation technique only so long as the source program is in a debug status. A purge function is used to return space to the operating system whenever a program is debugged to the programmer's satisfaction. Another INITIAL compilation will return the program to a debug status by regenerating all data necessary for another PARTIAL recompilation.

A PARTIAL recompilation is requested whenever modifications are to be made to the program source language. Rather than perform an entire program recompilation, source language modification requests are input to a PARTIAL recompilation, which operates as follows:

1. **Set-Up Phase**

   The set-up phase is designed to prepare the way for a PARTIAL recompilation. Its functions include:

   a. Read source language modification requests from the on-line CRT unit or from an "alter card" deck. These requests specify source language statements to be added, deleted, or replaced from the source language program.
   
   b. Build an update file which contains all information needed to update the source language file.
   
   c. Set indicators within the SNODCT table to indicate the presence of insertion or deletion requests and to identify source statements which must be recompiled during the PARTIAL recompilation. The number of such statements depends upon the type of source modification requests made.
   
   d. Construct object code modification tables which indicate the points at which machine executable instructions are to be inserted or deleted from the object code program, based upon the source code modification requests and the appropriate SNODCT data.

2. **Generator Phase**

   Unlike the generator phase in the INITIAL mode, the generator during a PARTIAL recompilation processes only a limited number of the statements comprising the source program. All new statements are processed, as well as any statements flagged for recompilation by the set-up phase. Dictionary entries are established for all new or recompiled data declarations and statement labels. New (temporary) intermediate data generated as a result of compiling new statements or of recompiling old statements is saved. A new (temporary) version of the SNODCT table is generated for new statements compiled as well as for old statements recompiled.

3. **Translator Phase**

   The translator phase operates exactly the same during a PARTIAL recompilation as during an INITIAL compilation, with the following exceptions:

   a. Intermediate object code table entries are generated for declarative data based upon those dictionary entries which are new or which resulted from a recompilation of old statements.
   
   b. Intermediate object code table entries are built for imperative data based upon entries of the new (temporary) intermediate analysis tables rather than the permanent version.

4. **Assembly Phase**

   The assembly phase during a PARTIAL recompilation operates exactly the same as during an INITIAL compilation except that in the former case a new (temporary) object code program is generated based upon the compiled and recompiled statements.

5. **Reconciliation Phase**

   The reconciliation phase is divided into two parts; object code reconciliation and source code reconciliation, as follows:

   a. The object code program generated during the INITIAL compilation or during the previous PARTIAL recompilation is modified based upon
the object code modification tables generated during the set-up phase and the object code formed by the assembly phase of the current PARTIAL recompilation. Appropriate object code program registers are deleted, new object code inserted, and address relocation performed. The resultant object code program now embodies all requested source language modifications.

b. New source language records from the update file created by the set-up phase are combined with the original source language statements to produce an updated version of source language. The SNODCT table and intermediate analysis tables are restructured based upon the new (temporary) versions and upon the source language modification requests in the update file. All program files and intermediate data are now in such a state that another partial recompilation could be performed.

One of the major considerations in performing a PARTIAL recompilation is to determine, based upon the type of source modifications made, exactly which source statements must be recompiled. For certain types of modifications, a PARTIAL recompilation may not be possible. For example, in the subject compiler an INITIAL compilation must be performed if the organization of the environment (data declaration area) is modified via the JOVIAL OVERLAY statement.

Ideally, only new source language records would be compiled during a PARTIAL recompilation, along with all statements which contained undefined data references or statement label references during the INITIAL compilation (or previous PARTIAL recompilation). Although many PARTIAL recompilations may result in the processing of such a limited number of statements, many types of source language modifications necessitate that unchanged but related statements also be recompiled. If a very large number of source language modifications of this type are requested, an INITIAL compilation might be better, i.e., faster than a PARTIAL recompilation. These determinations are made by the set-up phase of the compiler based upon source modifications requested and upon the content of the SNODCT table.

When the SNODCT table is constructed by the generator phase of the compiler, certain source language statements are "linked" together by setting switches within appropriate SNODCT entries. If any change is made to a statement within a linked group, or if new statements are inserted into a linked group, all statements within that group must be recompiled during a PARTIAL recompilation. Statements which are linked together for recompilation purposes in the subject compiler include the following:

1. Logically related data declarations, such as table entries or entries within a certain array.
2. Certain program loops (such as the JOVIAL "for loop" or FORTRAN "do loop") and conditional statements.
3. Certain program subroutines.

A change to declarative data necessitates that references to it be recompiled during a PARTIAL recompilation since it is possible that the modifications to the data declaration should result in different object code for processing the data, such as in changing a data definition from floating point to integer or vice versa.

Also, whenever, a statement which contains a statement label and is located within a linked group is recompiled, statements which reference the label must be recompiled since the relative object code address of the statement label is apt to change. Fortunately, this does not carry on forever, that is, such recompiled statements do not cause other statements which reference tags to them to be recompiled since only the address reference of the secondarily recompiled statement changes.

Admittedly, some redundancy still exists within the current version of the compiler. For instance, it should not always be necessary to recompile all logically related data declarations and their associated references whenever a small portion of the declaration (such as a table or array entry) is modified. Nor should it always be necessary to recompile an entire program subroutine whenever a portion of the subroutine is modified. This redundancy is the result of an attempt to keep modifications to the current version of the compiler relatively straightforward. By sophisticating the set-up and generator phases even less redundancy would exist during a PARTIAL recompilation.

TIMINGS

The partial recompilation technique was tested and timings made on five JOVIAL programs of various length. The timing figures set forth for these programs were obtained by using a high speed drum unit as the mass storage device. Compiler program loading times were subtracted from the actual elapsed times, since all JOVIAL compiler programs in the subject system are loaded from magnetic tape. Thus, timing comparisons tend to reflect time spent during the actual compilation process.

Although the secondary storage requirements given
for the five test programs are representative of the requirements of the subject compiler, it must be pointed out that these requirements could be reduced by employing more efficient data storage techniques than those currently used by the compiler. For example, 80 characters of storage is currently required for each line of source language, even though the line may contain considerably fewer than 80 characters of information. Optimization of secondary storage usage should result in storage requirements at least 20 percent less than those given in this report.

For comparison purposes, an INITIAL compilation was compared to a PARTIAL recompilation during which a single insertion into the source language was made. Partial recompilations which cause many additional source statements to be recompiled could be expected to take longer than the cases presented here. Usual debug outputs, such as source and assembly listings, were suppressed during both the INITIAL and PARTIAL runs. The accompanying graph represents “percentage of time saved” by performing a PARTIAL recompilation rather than an INITIAL program compilation.

<table>
<thead>
<tr>
<th>TEST PROGRAM 1</th>
<th>PARTIAL recompilation</th>
<th>21 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of statements</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Secondary storage requirements</td>
<td>24,266 characters</td>
<td></td>
</tr>
<tr>
<td>INITIAL compilation</td>
<td>21 seconds</td>
<td></td>
</tr>
<tr>
<td>PARTIAL recompilation</td>
<td>9 seconds</td>
<td></td>
</tr>
<tr>
<td>Percent time saved, PARTIAL vs. INITIAL</td>
<td>57 percent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST PROGRAM 2</th>
<th>PARTIAL recompilation</th>
<th>21 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of statements</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>Secondary storage requirements</td>
<td>73,350 characters</td>
<td></td>
</tr>
<tr>
<td>INITIAL compilation</td>
<td>81 seconds</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST PROGRAM 3</th>
<th>PARTIAL recompilation</th>
<th>146 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of statements</td>
<td>468</td>
<td></td>
</tr>
<tr>
<td>Secondary storage requirements</td>
<td>115,780 characters</td>
<td></td>
</tr>
<tr>
<td>INITIAL compilation</td>
<td>26 seconds</td>
<td></td>
</tr>
<tr>
<td>PARTIAL recompilation</td>
<td>7 seconds</td>
<td></td>
</tr>
<tr>
<td>Percent time saved, PARTIAL vs. INITIAL</td>
<td>75 percent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST PROGRAM 4</th>
<th>PARTIAL recompilation</th>
<th>216 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of statements</td>
<td>685</td>
<td></td>
</tr>
<tr>
<td>Secondary storage requirements</td>
<td>160,417 characters</td>
<td></td>
</tr>
<tr>
<td>INITIAL compilation</td>
<td>31 seconds</td>
<td></td>
</tr>
<tr>
<td>PARTIAL recompilation</td>
<td>8 seconds</td>
<td></td>
</tr>
<tr>
<td>Percent time saved, PARTIAL vs. INITIAL</td>
<td>82 percent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST PROGRAM 5</th>
<th>PARTIAL recompilation</th>
<th>296 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of statements</td>
<td>884</td>
<td></td>
</tr>
<tr>
<td>Secondary storage requirements</td>
<td>203,631 characters</td>
<td></td>
</tr>
<tr>
<td>INITIAL compilation</td>
<td>36 seconds</td>
<td></td>
</tr>
<tr>
<td>PARTIAL recompilation</td>
<td>9 seconds</td>
<td></td>
</tr>
<tr>
<td>Percent time saved, PARTIAL vs. INITIAL</td>
<td>88 percent</td>
<td></td>
</tr>
</tbody>
</table>

CONCLUSIONS

The technique of partial recompilation seems to present a means of reducing most of the redundancy associated with program recompilations. If implemented into a compiler which operates in a time-sharing environment, this time saving should result in improved terminal response time, particularly to the user attempting to debug a source language program conversationally.

The primary debugging advantages afforded by the incremental compiler and the interpreter are available, i.e., local syntactic errors may be corrected from the terminal as they are encountered.

The major advantages of a compiler with the partial recompilation capability are:

1. A programmer need not be concerned with program object code, since a partial recompilation can be performed in less time than a programmer could determine and make object code patches to an erroneous program. Programmers should be encouraged to perform source language corrections via a partial recompilation as soon as errors are detected.

From the collection of the Computer History Museum (www.computerhistory.org)
2. The need for sophisticated debugging packages is lessened since debugging commands and console typeouts may be entered at the source language level via a partial recompilation.

3. The partial recompilation technique reduces the burden (except for mass storage requirements) of the system upon which the compiler operates, since a minimum amount of time is spent on complete program recompilations.

The major disadvantages of a compiler with the partial recompilation capability are:

1. Due to the additional general housekeeping functions necessary, the construction of the compiler itself is necessarily more complex than for a compiler which does not offer the capability.

2. Additional mass storage requirements are imposed upon the system to accommodate the additional data which must be generated and saved in order to allow for a partial recompilation.

ACKNOWLEDGMENTS

The authors are grateful to their contract monitor, Captain Andrew M. Kobziar of the U.S. Air Force's Rome Air Development Center (RADC), for his very valuable contributions to the design and implementation of the partial recompilation technique. Acknowledgment is also made to Mr. Albert J. Kreutzer for programming assistance and to Messrs. Jack Rusman and Robert T. Strain for their initial guidance and inspiration.

REFERENCES

1 P V WATT
Generating and debugging programs at remote consoles of the PDP-6 time-sharing system
Proc Third Australian Computer Conference

2 H KATZAN JR
Batch, conversational, and incremental compilers
Proc SJCC 1969

3 K LOCK
Structuring programs for multiprogram time-sharing on-line applications
Proc FJCC 1965

4 J L RYAN R L CRANDALL M C MEDWEDEFF
A conversational system for incremental compilation and execution in a time-sharing environment
Proc FJCC 1966