Published: 12/07/67

Identification

A primer on EPL-compiled object code
D. B. Wagner

Epigraph

Thus the unfacts, did we possess them, are too imprecisely few to warrant our certitude, the evidencegivers by legpoll too untrustworthily irreperible where his adjugers are semmingly freak threes but his judicandees plainly minus twos. Nevertheless Madam's Toshowus waxes largely more lifeliked (entrance, one kudos; exits, free) and our notional gullery is now completely complacent, an exegious monument, aerily perennious.

- James Joyce

Purpose

The BN.10 Sections are a kind of overview to the various other BN Sections concerning the code produced by EPL. The approach is more tutorial than precise, and it will be noticed that in some cases the code compiled by EPL is considerably more bizarre than indicated here. This is a natural consequence of the desire (absent in other BN Sections) to avoid frightening the reader away.

Discussion

The present Section describes in general terms the overall structure of an EPL-compiled program and then points out some of the "hot spots" - places in which the code produced by EPL is more wretched than usual or where clever programmer decisions can make a great difference in the efficiency of the compiled code.

In Section BN.9.00 J. F. Gimpel makes an important point:

> It would be unwise to base the discussion on the idiosynchrosies of any particular version of the compiler. Not only would the discussion be obsolete with the introduction of a new compiler, but the programs written using that advice would be sprinkled with obsolete glitches.

This is very good advice, which I concur with more than the rest of this Section would lead one to believe. But it must be balanced against the necessity of making Multics work. Since it does not appear that there will be any compiler better than EPL for at least a year, it will be necessary to keep EPL's bad habits in mind while writing programs.

The Structure of an EPL Object Program

The diagram of figure 1 shows some of the pieces of an EPL object program and some of the interconnections between them. We have ignored begin-blocks, internal procedures, and on statements. A solid arrow represents any transfer of control which can be considered a call and has a guaranteed return. A broken arrow represents an ordinary transfer of control, a broken non-arrow represents a data reference, and a zap represents a call caused by a fault.

The pieces of the program will not necessarily be found in the order shown in the diagram. Each piece is built up in little chunks using the "multiple location counters" feature of EPLBSA (see BN.8.01), and where EPLBSA puts each piece is its own business. Below we will briefly describe each piece of the program. Concrete examples of what we are talking about will be found in BN.10.01 and BN.10.02.

The prologue contains any code which should be executed before the main sequence of code. Among the jobs assigned to the prologue are:

1. Evaluating the extents of adjustable automatic items and allocating storage for them.

2. Establishing initial values for items declared automatic initial.

3. Setting up specifiers for automatic and based items which require them.

4. Calling condition to establish the epilogue, if necessary. (Epilogues are discussed below.)

5. Many miscellaneous things: for example performing the verify option if it was specified.

A discussion of the prologue appears in BN.9.02 and may be of interest. In EPL the prologue always has the form of a subroutine called with a tsx0 instruction. It is called at every entry point to the procedure.

A program may use one or several different pieces of static storage which must be grown at run time. Each external static variable or aggregate has a separate block of grown storage, and in addition the program needs a block of storage to hold all internal static storage declared in the program and all specifiers which happen to be needed for any static data. Growing of storage at run-time is done through use of the library procedure datmk_ in conjunction with the "trap-before-link" feature of the linker. See BP.4.01 for more detail on the operation of datmk_. Datmk_ gains control on first reference to a needed block of static storage, grows the storage, and if necessary passes control to a piece of the procedure (referred to in the diagram as the static initializer) which initializes the storage. There are two jobs this procedure might have:

1. Establishing initial values for items declared static initial.

2. Setting up specifiers for static data which need them. These specifiers are all kept in the same grown storage block with internal static variables, and initialized with them.

Much of what is said in BN.9.02 about the prologue also applies to the static initializers.

The epilogue contains the cleanup operations which must be performed before the procedure becomes inactive. The two jobs of the epilogue are:

1. Releasing the storage occupied by automatic varying strings. (See BP.2.01.)

2. Reverting on-units, (See BN.5.02.)

See BP.3.00 (not BN.5.01) for a fuller discussion of epilogues.

There are two different ways in which the epilogue may be invoked: at the return statement, or in the course of a non-local go to. (The diagram shows that our procedure called some other procedure which called the unwinder to execute a non-local go to to a dynamic ancestor of our procedure.) In order to make it possible for the unwinder to invoke the epilogue, the prologue includes a call to the system procedure condition which keeps a stack of epilogues. The epilogue compiled by EPL is very cleverly worked out so that it may be either called or transferred to.

The compiled code uses various special (and sometimes very peculiar) subroutines compiled in at the end of the procedure. The diagram shows, for example, the call to .sv which is a single instance of the standard Multics save sequence. Others perform the mod function, do subscripting, calculate shifts, convert from floating to fixed, and on and on. Section BN.3.02 is useful in deciphering what these subroutines do, but unfortunately the version dated 2/24/67 is at this writing a bit out of date, and many subroutines compiled by EPL are not described there.

Evaluating EPL Object Code

It cannot be stressed too strongly that EPL is so complicated that it is not possible to be sure ahead of time of the consequences of various courses of action. If a programmer is seriously interested in optimizing object code by source program decisions, he must be prepared to iterate: make a change, see the results, change again (or change back)....

Furthermore, and most important, a body count tells very little about whether you are winning or losing. Some instructions are more strategically placed than others. Looking at the size of a compiled program is certainly easy, but the ease of counting instructions and the difficulty of making timing tests has led people to assume a proportionality between size and speed which simply does not exist. Section BN.10.01 shows two programs which do the same thing and are roughly the same size: one is four times faster than the other.

An easy way to do timing tests in 6.36 is to use the 645 interval timer. This is a 24-bit hardware register which counts (depending on the setting of a switch on the processor panel) either memory references or ticks of an internal 64 kc. clock. A memory reference is approximately 1 microsecond, and a clock tick is approximately 15 microseconds. At this writing, the switch is always set for clock ticks, but this is subject to change.

The interval timer is referenced using the machine instruction stt. A procedure to get the timer reading from an EPL program is being placed on the Multics Segment Library. To use it:

```
dd timer external entry returns (fixed bin (24));
.... = timer;
```

There is one problem with using the interval timer: while the program is running, GECOS may steal some time in order to print on-line listings. Thus there may be considerable error in the timings taken using this method. If this turns out to be a problem, the 645 simulator will help. (See BE.5.02 for how to use it. Note that the simulated interval timer always counts memory references.) Unfortunately the simulator is desperately slow, taking about 30 minutes to simulate 1 second. A few simulator jobs submitted in a day can completely wreck turn-around time for everyone using the 645.

Hot Spots: Long and Short Strings

Given the brief discussion of an EPL compilation above, we now discuss a few of the places where EPL-compiled code is especially inefficient, or where minor programmer decisions can have major effects. The first and perhaps most important of these "hot spots" concerns string manipulation. Note that everything said below applies equally to bit-strings and character-strings.

Non-varying non-adjustable strings with length known and less than or equal to 36 bits are called short strings. Others, including all varying strings, are called long strings.

Nearly all operations on long strings are performed through calls to the run-time routines described in BN.7.09. Not only are these routines very general and slow, but their use means that the strings need specifiers and dope, which makes the prologue bigger. Furthermore varying strings (as presently implemented) must be initialized and terminated through calls to the library procedures varst_$zero and varst_$clear, described in BN.7.02. Thus varying strings place an additional burden on the efficiency of the program.

Operations on short strings are normally performed in-line and tend to be much faster than operations on long strings (perhaps by a factor of 15). Whenever possible programs which manipulate strings should be designed to use short strings, although of course it is clear this is not always possible.

Section BN.9.01 discusses some of the efficiency considerations involved in using short strings, especially within aggregates. One important consideration mentioned there is that a short string parameter is difficult to access (perhaps 10 times slower than an aligned automatic string) so that a program which accesses such a parameter more than once should make a copy of it instead. To make it easier for the programmer to detect cases like this, the EPL compiler places in the EPLBSA file the comment "IDIOTIC" next to every access to a short bit-string parameter (as well as in some other places: see below).

Section BN.10.01 contains an extended example of the kinds of things that happen in EPL string manipulation.

Hot Spots: Planning of Aggregates

The planning of aggregates is one of the very tricky and yet very important hot spots of EPL. Sections BN.9.01 and BN.9.01A are both quite readable and devoted primarily to this subject, and we cannot possibly say much here except to recall some of the major conclusions of those Sections:

1. Aligned aggregates are much faster to access than packed ones, so much so that the programmer has almost no reason for using packed aggregates.

2. Arrays with one-word elements are easier to access by close to a factor of 10 over nearly equivalent arrays which have other element sizes.

3. Some rules are given in BN.9.01 showing how adjustable aggregates should be laid out. Usually a properly Gimpelized structure will have its elements laid out in the following order:

   a. Pointers, labels, and double-precision arithmetic variables.
   b. All other non-adjustable items.
   c. An array with only the first upper bound adjustable.
   d. All other adjustable items.

4. Two fairly complicated concepts are defined in BN.9.01, synchronous and idiotic. To help the programmer find instances of certain kinds of inefficiencies, the EPL compiler places the comments "NOT SYNCHRONOUS" and "IDIOTIC" in the EPLBSA code to mark accesses to aggregate elements which would be more efficient if the aggregate were planned more carefully.

Some examples of the planning of aggregates will be found in BN.10.01 and BN.10.02. Another example is the Active Process Table described in BJ.1.01. BJ.1.01 contains a rather good discussion of the considerations that went into the design of the APT.

Hot Spots: Epilogues

An epilogue is needed in every program which contains an on-statement or an automatic varying string. In such a program there must be at least two special calls, one in the prologue to establish the epilogue for the unwinder, and one in the epilogue itself to get rid of the unwinder's record of the epilogue. So if possible varying strings should be made static and on-statements should be eschewed.

To determine quickly whether a given program uses an epilogue check toward the end of the EPLSBA listing for the symbols CLEAN.UP, CLEAN.P, and BEGIN.E.

Occasionally the programmer will be mystified by the occurrence of an epilogue in a program which has no on-statements and no declarations of varying strings. The curious epilogue in such a case will undoubtedly be because of a varying string temporary generated by the compiler. Such temporaries can be tracked down by looking for references to <free_>|[free_] in the code. After the offending statement is found, the programmer can fiddle and try to get rid of the varying temporary.

Hot Spots: Fancy Do-Statements

Imagine the worst code possible for the statement:

```
do i = 2,3,5,7,11,13,17,19,23,29,31;
```

The reader who has never tried a statement of this form will undoubtedly have guessed low: in a test EPL compiled 3 1/2 pages of EPLBSA code for this statement.

The word: eschew do statements with multiple specifications.

However EPL special-cases the commonly used forms of the do statement so in general it is not necessary to be too concerned about them.

Hot Spots: The Bug in the if Statement

A bug which has existed in EPL since its earliest days causes if's to be much slower than necessary in some cases. Consider the statements:

```
dcl fizzy external entry returns (bit(1));
if fizzy then go to indigestion;
```

The bug in EPL causes fizzy to be called twice, which is very sad.

In general if the major operator in the expression is not a relation then the entire expression is compiled twice. There is some sign that this nonsensical bug may go away in the next two months or so.

Hot Spots: Miscellaneous

A few additional problem areas are:

1. Begin - blocks cost something and should not be used without a reasonably reasonable reason. Part of the cost lies in the fact that accesses in the block to automatic data in containing block take an extra instruction each. See also BN.9.01.

2. Adjustable data is implemented wretchedly. Unfortunately where it is used there is seldom a substitute. However see BN.10.02 for some hopeful suggestions.

3. A use of the substr built-in function usually requires two calls to run-time routines. (See BN.7.05.) This will eventually change but not for quite a while. In the meantime, for those who cannot wait, various fancy mismatched declarations can often be used to speed things up. See BN.10.01 for one example of this kind of game-playing. (However note that the use of mismatched declarations and other implementation-dependent constructions is a Bad Thing: see BN.10.01).

Non-Hot Spots

It is not generally necessary to worry about the following statements. This is not to say that they are all terribly efficient, but that if they are inefficient they are inherently so and only a change in the philosophy of the program could possibly improve the situation.

allocate

call

do (except as mentioned above)

entry

free

go to

else

on

procedure

return

revert

signal

For anything not mentioned here it would be wise to take a glance at the EPLBSA code produced.

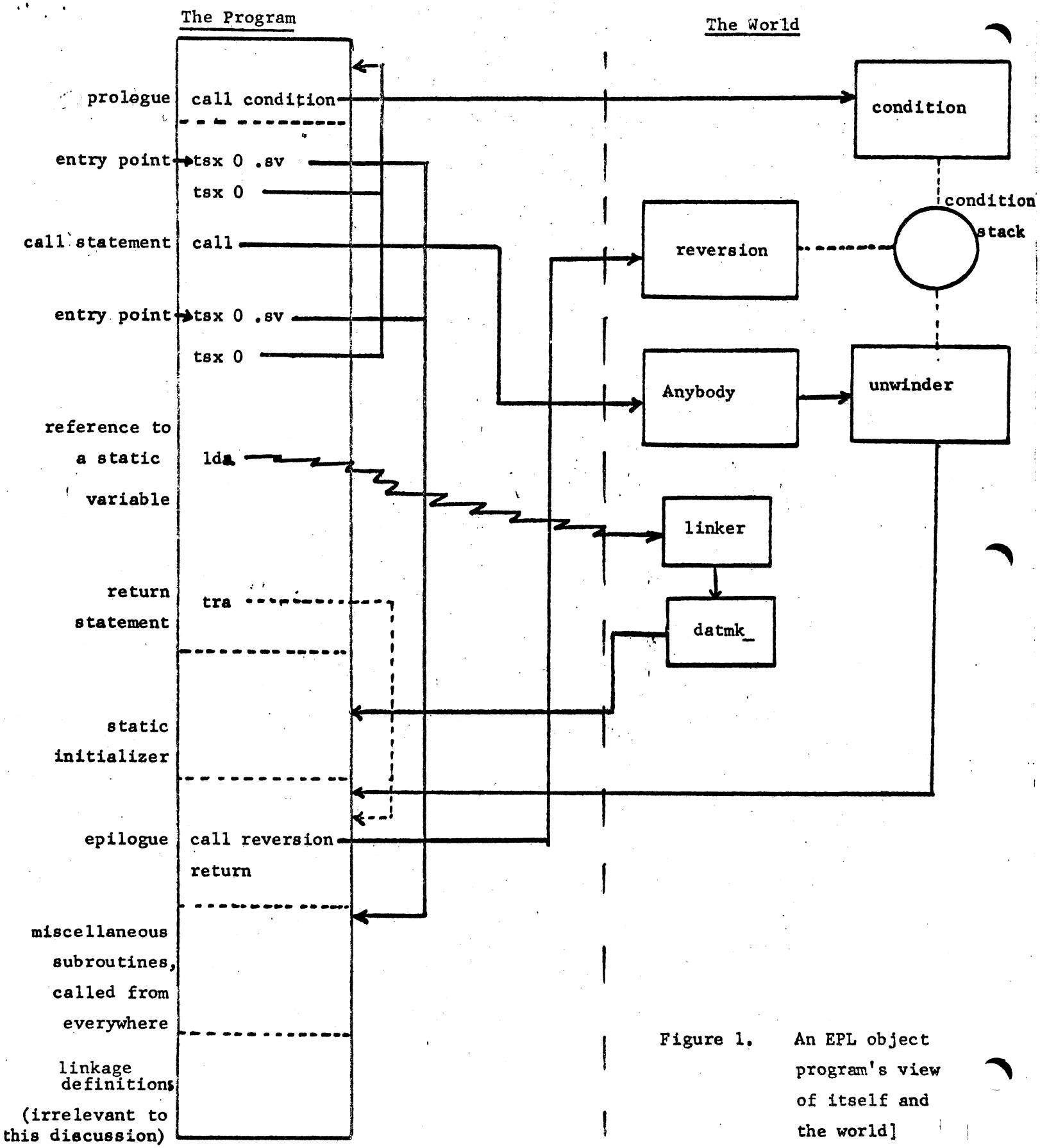

Figure 1. An EPL object program's view of itself and the world]