LITTLE Newsletter No. 20

November 16 1972
David Shields

Remarks on structure of SETL Run-time Library

We remark on some of the properties of the LITTLE coded run-time library for SETL (SRTL). Based on a casual reading of the routines currently available, these remarks are intended to describe the 'feel' of the code, especially from the optimiser's point of view, and to indicate which optimisations in LITTLE may have the most immediate payoff for SRTL.

The code section examined contained about 2500 executable LITTLE statements. There were about 60 subroutines (SUBR's) and about 30 functions (FNCT's), and about 80 CALL statements.

Basic blocks typically contained three to five statements, for a total of about 800 basic blocks. As expected, the handling of types accounted for most of the program flow,- the typical subroutine takes the form

```
SUBR  XXX;
GO BY type-of (object) (intlab,reallab, setlab,...);
/INTLAB/...;  GO TO DONE
...
/SETLAB/...;  GO TO DONE;
```

many of the blocks generated by type-branching were null, i.e., of the form

```
/SETCASE/;
/NEXTCASE/ ...
```

The following situation occurs perhaps a hundred times:

```
IF (c) GO TO  TRUECASE;
GO TO FALSECASE;
/TRUECASE/ ...; GO TO NEXT;
/FALSECASE/ ...;
```

where the /TRUECASE/ block has the IF-statement as its only predecessor. The preferred code sequence is

```
IF (.NOT. c) GO TO FALSECASE;
...TRUECASE code...; GO TO NEXT;
/FALSECASE/
```

Almost all of the subprograms have no arguments since the SRTL routines use their own conventions for passing arguments. Thus the following pattern is common

```
MEMORY (c) = arg-value $ c is constant
CALL XXX;
....
```

where XXX is of the form

```
SUBR XXX;
IF ( MEMORY (c)  ) .....
```

Thus, on entry to many of the routines, input arguments could have their input values available in registers; assuming we perform the suggested global optimisation between subroutines.

The need for optimisation of array references is self-evident. The great majority of memory references have the form

```
STORAGE (1000 - constanta +  constantb)
```

due to the manner in which arguments are passes between routines. Note that since STORAGE is a global variable we should be able to compile

```
= STORAGE + 887
```

so that the loader can do subscript reference.

The following uses of the field-extractor .F. are common:

a) .F. 52,5, x = .F. 52,5,y
b) .F. 18,17,x = .F. 18,17,y
. .F. 35,17,x = .F. 35,17,y

c) IF (.F. 52,5 x .EQ. 10) GO TO...

d) STORAGE(.F. 1,17, x) =

The extractor will in general compile as follows

generate-mask (by load or perhaps special code for certain masks)

shift-input
and input with mask

Example a) above contains a repeated instance of same field. Thus compiler should avoid duplicate generation of mask, either by constant propogation, or by code-generator for extractor.

Example b) is instance of "parallel" field assignments, in which no shifts are necessary, and only one mask need be used.

Example c) is instance of an optimisation suggested by in LITTLE newsletter 18, in that inputs to conditionals need not always be reduced to final boolean form.

In example d), the subscript is a "pointer". Note that LITTLE does not allow pointers as separate data type, and that references of this form result when the user manages his own memory.

In summary, the LITTLE code for the SRTL has the following major characteristics

-- few if any formal arguments
-- small basic blocks, including many null blocks
-- many references to global array with subscript an constant expression known at compile time
-- extraneous branches due to manner in which macros used to generate LITTLE code expand, so that simple reodering of code would be quite helpful
-- many "pointer" variables, ie, variables used to index STORAGE, which should be kept in B- registers, particularly for loops
-- a substantial amount of address arithemtic, or computations whose result is used as subscript
-- substantial redundancy in use of field-extractors especially those used in address-arithetic

The suggested LITTLE optimisation efforts may be conveniently devidied into those with an immediate payoff and which might be implemented fairly easily, and those which are more difficult to implement. The immediate optimisations are those suggested by Jack Schwartz in Newletter 18, i.e. constant propogation, good generation of constants, better basic block processor, and better register allocation within blocks. However, the small size of most basic blocks indicates that over-elaborate basic block processors would not be of much help now.

The longer term efforts should concentrate on global flow analysis within routines, the isolation of address arithmetic, especially the machine-dependent parts. Perhaps the major obstacle in implementing the machine-dependent optimisations is the structure of the current assembler in the LITTLE compiler; it is suggested that the assembler should be "isolated" from the parser by having them communicate through files.

Also, having the compiler produce code for a hypothetical LITTLE machine which is then assembled in a machine-dependent way, seems much better in the long run than having the compiler produce absolute load modules.

In a subsequent newsletter we intend to present some of the machine code currently produced by the LITTLE compiler nor the SRTL source. Analysis of this code could indicate more precisely the payoff of the optimisations suggested. Intuition suggests that it should be possible to reduce memory requirements and running time for SRTL by 40 to 60 percent by impoementing just the optimisations mentioned above.