stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 112/79 JUNI

P. KLINT

LINE NUMBERS MADE CHEAP

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS Subject classification scheme (1980): 68A10

ACM-Computing Review-categories: 4.42, 4.10

Line numbers made cheap *)

by

Paul Klint

ABSTRACT

A technique is described for run-time line number administration to be used for implementations of high-level languages. Under suitable circumstances, this method requires absolutely no overhead, in either time or space, during execution of the program.

KEYWORDS & PHRASES: Line number administration, diagnostic messages, abstract machine code.

*) This report will be submitted for publication elsewhere.

1. INTRODUCTION

This note describes a new technique (henceforth called LMC: Line numbers Made Cheap) for run-time line number administration to be used by implementations of high-level programming languages. Such an administration is needed for determining the source program line number when a run-time error condition occurs. This line number can then be used in a diagnostic message describing the kind and origin of the error.

The simplest method for maintaining the line number at run-time is to introduce a system variable "LN" (for Line Number) and augment a given source program with statements that assign appropriate values to this variable. For example, the program fragment

```
(1)  x := 3;
(2)  if a > b then
(3)       y := 4.2
(4)  else
(5)       z := 0.002
(6)  fi;
(7)  while q < r
(8)  do
(9)       q := q * x + y
(10) od
```

is transformed into

```
     LN := 1;
(1)  x := 3;
     LN := 2;
(2)  if a > b then
          LN := 3;
(3)       y := 4.2
(4)  else
          LN := 5;
(5)       z := 0.002
(6)  fi;
(7)  while LN := 7; q < r
(8)  do
          LN := 9;
(9)       q := q * x + y
(10) od
```

If an error condition occurs during the execution of the transformed program, say arithmetic overflow in the statement q := q * x + y, then the value of LN is equal to the line number of that statement in the original program. This method increases the execution time and the size of the resulting program. It has been observed [1] that upto 25% of the generated code may consist of instructions devoted to line number book-

keeping. Such an observation leads to (undesirable) compiler options to suppress the generation of line number information in the code.

However, these costs can be reduced if the compiler performs a modest semantic analysis of the source program and determines which statements may cause run-time errors. It is sufficient to prefix precisely these statements with line number instructions. In [2] such an analysis is described for Algol 60.

A completely different method, used in an implementation of Algol-W [3], is to construct (at compile time) a separate table which relates addresses in the generated code to source program line numbers. When an error occurs at run-time, the current value of the program counter is used as an index in this table and this yields the line number of the statement currently being executed. This method does not affect execution time or program size, but has the disadvantage that the table with line number information is an entity disjoint from the actual object program. This increases the complexity of the overall system.

A third method, which is the subject of this paper, is to combine the line number information with frequently occurring operations in the object program. Then, the line number corresponding to a given machine instruction location can be determined by simply scanning the object program from the beginning and accumulating the line number information. This method will now be considered in more detail.

2. THE LMC TECHNIQUE

We assume that a high-level language compiler produces low-level code for an abstract machine. This approach is gaining in popularity, since it allows the production of portable compilers. The abstract machine code may either be interpreted or assembled to executable machine code. This distinction is not essential, but it is easier to incorporate the LMC technique in interpreter-based systems.

Three conditions must be satisfied to make the LMC technique a viable alternative for existing line number administration methods:

- The value of the current line number is not needed very frequently, and therefore it is acceptable if determining the line number requires a fair amount of computation. This is the case for line numbers in diagnostic messages and program traces, but not for program-defined traps that depend on a particular value of the current line number.

- The instruction set of the abstract machine can be modified to handle the line number administration. When used with real machine code, there must be unused fields or opcodes in which the line number information can be encoded.

- Individual operations in the abstract machine code can be inspected.

If these three conditions are met, one can expect the following benefits from application of the LMC technique:

- The line number administration hardly increases the size of the abstract machine program.

- Run-time maintenance of line numbers does not impose a penalty in execution speed.

- The accuracy of the line number, as computed by the LMC algorithm, can be determined by the algorithm itself. Whether that accuracy is acceptable or not depends on the specific application and the integration of the LMC model in the abstract machine code.

The LMC technique is based on two ideas. The first idea is to associate the line number administration with a frequently used abstract machine operator, that occurs in the translation of (almost) every high-level language statement. This operator must have the property that its order of appearance in the abstract machine code is the same as its appearance in the high-level language program. This excludes, for example, the assignment operator (and in general any right associative operator) from being used for this particular purpose. We will consider an abstract machine with stack oriented architecture and associate the line number information with the VOID operator, which removes the top stack element. The associated information consists of the line number

increment since the previous VOID operator in the abstract machine code. This information is based on the static occurrence of VOID operators in the code and not on the dynamic behaviour of the program.

The second idea is that line numbers need not be maintained at run-time at all! When the value of the current line number is required, inspection of the abstract machine code allows that value to be reconstructed. It turns out that the current value of the (software) program counter and the static information associated with VOID operators is sufficient to compute the line number. One only needs to scan the abstract machine operations and accumulate the line number increments on the fly. A lower bound (LB) on the line number is the total of increments accumulated before the value of the program counter is reached. An upperbound (UB) is the lowerbound minus one plus the first non-zero increment that occurs after the value of the program counter. These rules need some refinement for the case that the operator to which the line number information is attached can cause a run-time error itself. These refinements are straightforward and will not be considered here.

Consider the example:

```
(1)     x := y + 1;
(2)     z := 2;
```

with translation:

```
LN LB UB  CODE        COMMENT

1  1  1   LOADV   y   stack value of variable y
1  1  1   LOADC   1   stack constant 1
1  1  1   ADD         replace 2 top elements by their sum
1  1  1   STORE   x   store top element in variable x
1  1  1   VOID    1   remove top element; line increment 1
2  2  2   LOADC   2   stack constant 2
2  2  2   STORE   x   store top element in variable x
2  2  2   VOID    1   remove top element; line increment 1
```

The column labelled with LN gives the real line number, the columns labelled with LB and UB give the lowerbound and upperbound as computed by the LMC algorithm. In the examples we will assume that initially LB=1 holds, and that end-of-line is associated with the expression on the line just being ended and not with the expression on the next line. The real line number LN must satisfy:

$$LB \leq LN \leq UB$$

If LB and UB are equal, the exact value of the line number has been com-

puted. If not, the line number has a value in the interval [LB,UB]. Hence, the inaccuracy of the value computed for the line number is always known.

If we rewrite the first line of the above example to

```
(1)     x :=
(2)          y +
(3)              1;
```

the generated code and associated line numbers would look like:

```
LN LB UB  CODE        COMMENT

2  1  3   LOADV   y   stack value of variable y
3  1  3   LOADC   1   stack constant 1
2  1  3   ADD         replace two top elements by their sum
1  1  3   STORE   x   store top element in variable x
3  1  3   VOID    3   remove top element; line increment 3
```

This illustrates the point that the abstract machine operators and the associated real line number do not need to appear in order of increasing line number.

In general, the line number increments are small, say less then 10. One can use this observation in the following way. Instead of introducing one operator

```
VOID increment
```

a series of operators can be defined for the frequently occurring special cases:

```
VOID0 = VOID 0
VOID1 = VOID 1
  ...
VOID9 = VOID 9
```

By introducing these new operators, no extra space for the increment is needed in the abstract machine code. Moreover, these special VOID operators can be interpreted as efficiently as the normal VOID operator. In fact, they are synonyms of the old VOID operator and can be treated accordingly. The additional information encoded in these synonyms is only used during the computation of the line number.

A final example illustrates the computation of line numbers when conditional flow of control is involved. The high-level language statement:

```
(1)     if p > 0 then
(2)              q := 3
(3)     else     r := 4; s := 5 fi
```

leads to the following code and line numbers:

```
LN LB UB  CODE          COMMENT

1  1  2    LOADV   p    stack value of variable p
1  1  2    LOADC   0    stack constant 0
1  1  2    JLE     L    jump to L on less then or equal
                        and remove top 2 elements from stack
2  1  2    LOADC   3    stack constant 3
2  1  2    STORE   q    store top element in variable q
2  1  2    VOID2        remove top element; line increment 2
2  3  3    JMP     M    jump to M
3  3  3 L:LOADC   4    stack constant 4
3  3  3    STORE   r    store top element in variable r
3  3  3    VOID0        remove top element; line increment 0
3  3  3    LOADC   5    stack constant 5
3  3  3    STORE   s    store top element in variable s
3  3  3    VOID1        remove top element; line increment 1
4  4  4 M:
```

3. EXTENSIONS

1 The LMC algorithm can also be used to keep track of statement numbers instead of line numbers.

2 The method can be extended to deal with problems raised by global optimizations, such as moving code out of loops and eliminating common subexpressions. The extension consists of introducing the additional instruction

```
ALINE N
```

to define an absolute line number N. When this instruction is encountered during computation of the line number, it has the effect of assigning the value N to both LB and UB. Moved code fragments must be surrounded with ALINE instructions to define line numbers corresponding to the line number of their original position in the code and to the line number of the code that follows. An additional ALINE is also needed to fill the gap in line number increments at the place where the code was moved from. Common subexpressions are handled by replacing all deleted subexpressions by ALINE instructions that take care of the line number increment caused by the deleted expression. Addition of ALINE instructions causes a (probably insignificant) increase of execution time and program size. In the worst

case, the optimized program including ALINE instructions is not better than the unoptimized program!

3 When it is impossible to add new instructions or employ unused fields in the machine code, one is forced to generate additional instructions for the line number bookkeeping. Even under such unfavourable circumstances, the LMC technique can be used to advantage. For example, on the PDP11 a compiler might generate dummy instructions of the form

```
MOV Rn,Rn
```

where Rn stands for one of the machine registers R0 through R6. The register number can be used as line number increment. This instruction occupies two bytes of code and requires 600 ns to execute. Compare this with the case that line numbers are maintained explicitly by means of instructions of the form

```
MOV #linenumber,LN
```

which require six bytes of code and approximately 1200 ns to execute.

On the CDC CYBER one uses dummy instructions of the form

```
SB0  Xj+Bk
```

to enforce certain alignment requirements. The (unused) register indices j and k provide a convenient way to encode a six-bit value for the line number increments.

On the IBM S/370, one can think of inserting operations of the form

```
BC   0,increment
```

This gives a four-bit field for the line number increment and a fairly low cost in execution time and program size.

4. REFERENCES

[1]

Hansen, P.B. & Hartmann, A.C., Sequential Pascal report, Information Science, California Institute of Technology, 1975

[2]

Kruseman Aretz, F.E.J., On the bookkeeping of source-text line numbers during the execution phase of Algol 60 program, in MC-25 Informatica Symposium, Mathematical Centre Tracts 37, 1971.

[3]

Satterthwaite, E., Debugging tools for high level languages, Software Practice and Experience 2(1972) 197-217.