technical contributions

The Best Simple Code Generation Technique for WHILE, FOR, and DO Loops

Forest Baskett
Los Alamos Scientific Laboratory *

ABSTRACT

This code generation technique for WHILE, FOR, and DO loops is simple to implement and usually results in the best loop code in the absence of flow analysis. Also the technique makes it possible to move code from inner loops without doing flow analysis and without ever moving code from a less frequently executed block to a more frequently executed block.

There are at least three things wrong with the following obvious code generation for WHILE, FOR, and DO (77) loops.

```
loop:    compute termination condition
         if condition then goto finis
         body of loop
         compute increment if any
         goto loop
finis:   rest of program
```

First, each execution of the loop requires the execution of two jump instructions--a conditional jump near the top of the loop and an unconditional jump at the bottom of the loop. The unconditional jump is unnecessary and could be quite expensive for some inner loops on some machines.

Second, if a machine is capable of parallel execution of independent instructions like a CDC 6600 or 7600, an IBM 360/91 or 370/195, or a CRAY-1, much of the opportunity for parallel execution of the computation of the loop termination condition and the body of the loop is lost by this style of loop code.

Third, if one wants to extend a simple basic block optimizer to do code motion from inner loops, this style of loop code requires execution frequency information to avoid moving code from a less frequently executed block to a more frequently executed block.

* Author's Address: Los Alamos Scientific Laboratory, P. O. Box 1663, Los Alamos, New Mexico 87545.

There is a better technique that solves all these problems. It has probably been described before but none of the recent books on compiler construction mention the technique. This technique is illustrated by the following code:

```
        compute termination condition
        if condition then goto finis
loop:   body of loop
        compute increment if any
        compute termination condition
        if not condition then goto loop
finis:  rest of program
```

More code is required by this technique although the increase is normally quite small. Now each execution of the loop requires execution of only one loop-associated jump instruction--the conditional jump at the end of the loop. Opportunities for parallel execution of the loop body and the loop control are now increased.

If a compiler distinguishes the loop label as being generated only for this type of loop construct, then an associated basic block optimizer will be able to discover a basic block that is an inner loop because it will begin with such a label and end with a conditional jump to that label. The basic block optimizer then can perform code motion from inner loops of this sort and always guarantee improvement of the code.

The position right before the label is a safe place to move code out the top of the loop. Code in that position is never executed if the loop is executed zero times. Likewise, the position just after the conditional jump at the bottom of the loop is a safe place to move code out the bottom of the loop. For example, a basic block optimizer concerned with doing good register allocation for basic blocks could move loads (as well as loop invariant computations) out the top of the loop and it could move stores out the bottom of the loop.

Even without the optimizations mentioned above, this code generation technique will improve loop code and it is easy to implement. In less than a day the author changed the FOR loop code generator of a PASCAL compiler for the CRAY-1 to use this technique, and the improvement in execution time was noticeable in many programs.