CS143
Compilers

Lecture 13
August 15, 2001

Code Optimization

- Target code produced by a code generator is far less efficient than what can be written by hand.
- The optimization phase of a compiler is devoted to improving the efficiency of the generated target code.
- While profiling a program on sample input can provide useful information for improving efficiency, a compiler must instead rely on heuristics to determine which optimizations to make.
- Optimizations arise largely from local transformations of code, as well as data-flow analysis.

Criteria for Code-Improving Transformations

- First and foremost, transformations must preserve the "meaning" of the original code.
- Second, transformations must improve the efficiency of the code to a measurable degree. The criteria may vary: in some cases maximum speed is desired, whereas in others minimum code size is preferred.
- Third, transformations must be worth the effort required to design and implement them. A compiler writer should not go to extraordinary lengths on a transformation that will have negligible impact.

Getting Better Performance

- Performance can be improved at all levels, from the source code to the final target code.
- Programmers can improve running time by selecting the most efficient algorithms.
- The front end of a compiler can apply generic transformations to intermediate code pertaining to loops, procedure calls, and address calculations to improve efficiency.
- The back end can improve performance through intelligent use of registers and instructions.

Optimizing Intermediate Code

Optimizing intermediate code, rather than target code, can be more beneficial because:

- Intermediate code expresses high-level constructs more explicitly, thus making them easier to optimize
- The optimization phase can be relatively independent of the choice of target language
- An additional optimization phase following target code generation can focus exclusively on platform-specific optimizations

Function-Preserving Transformations

- We have previously seen structure-preserving, also known as function-preserving, transformations that can be applied to code that can improve efficiency without changing the function, or structure, of the given code.
- Often these transformations of code are not available to the programmer because they pertain to functionality that is below the level of the source language.
- Examples of such transformations are common subexpression elimination and dead code elimination.

Copy Propagation

- Another useful function-preserving transformation seeks to limit the phenomenon of *copy propagation*, which arises from statements that simply assign a value from one name to another.
- Having multiple copies of a value, stored under different names, can increase the number of variables unnecessarily.
- Given a statement of the form **f** := **g**, use **g** in place of **f** wherever possible after the assignment. It may then be possible to eliminate this assignment altogether.

Loop Optimizations

- Typically, much of execution time is spent in inner loops.
- It can be very worthwhile to minmize the amount of code within an inner loop, even at the expense of increasing the amount of code outside of the loop.
- Three techniques for optimizing loops are:
  - Code motion, which moves inner loop code to outer loops when possible
  - Induction-variable elimination, which optimizes the use of several variables within a loop whose values change in lock step
  - Reduction in strength, which replaces expensive operations with cheaper ones

The dag Representation of Basic Blocks

- A directed acyclic graph, or dag, is a useful data structure for implementing both structure-preserving and algebraic transformations on basic blocks.
- A dag for a basic block is a directed acyclic graph with the following properties:
  - Leaves are labeled by unique identifiers, which may be variables or constants.
  - Interior nodes represent operators.
  - Interior nodes are optionally assigned names, since they represent computed values

Dag Construction

- The construction proceeds by creating nodes of the dag and assigning, or attaching, variables to the nodes. A variable *x* is attached to a node *n* if *n* represents the current value of *x*.
- For each three-address statement:
  - Obtain nodes for the operands, creating them if they don't exist. Attach the operands to the nodes.
  - Obtain a node *n* for the given operation with the given operands as children, creating it if it doesn't already exist
  - Let *x* be the result of the statement and *m* be the node holding *x*'s value. Detach *x* from *n* and attach *x* to *m*.

Applications of Dags

- Common subexpressions are automatically detected.
- We can determine which names have their values used in the block: they are those names for which a leaf is created.
- We can determine which statements compute values that could be used outside of the block. They are those statements x := y op z where x is still attached to the node that is found or created for that statement.

Loops in Flow Graphs

- In a source program, loops can easily be identified from the syntax of the language.
- In intermediate or target code, loops must be detected from the flow graph before they can be optimized.
- Loops can be detected by identifying *dominators* in the flow graph. A node d dominates another node m if every path from the initial node to m passes through d.
- An easy way to detect a loop is to find a *back edge* in the flow graph, which is an edge whose head dominates its tail.

Reducible Flow Graphs

- A *reducible flow graph*, intuitively, is a flow graph satisfying the property that every loop has a single entrance, through a node known as the *header*.
- In practice, nearly all source programs can be translated into code that can be represented by a reducible flow graph.
- In a reducible flow graph, every loop must contain a back edge, thus making loops easy to detect.
- Various optimization techniques are ideally suited to reducible flow graphs.

Global Data-Flow Analysis

- While local transformations can significantly improve performance of target code, a compiler needs global information about the actions performed in the source program in order to generate the most efficient code possible.
- Knowing when in the code variables are defined and used is essential for detecting when transformations can be applied both within and across basic blocks, as well as for optimal register allocation.
- The process of gathering such information is called *data-flow analysis*.