CSC 4181
Compiler Construction

Code Generation & Optimization

Code Generation

- Parser is modified to:
  - Output assembly language (or other low level) code for each statement
  - Use machine-specific knowledge
  - Perform simple or complex optimizations
  - Map statements to assembly code

Compiler Code Optimizations

- A Code optimizer sits between the frontend and the code generator.
  - Works with intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.
  - Does transformations to improve the intermediate code.

Optimizations provided by a compiler includes:

- Inlining small functions
- Code hoisting
- Dead store elimination
- Eliminating common sub-expressions
- Loop unrolling
- Loop optimizations: Code motion, Induction variable elimination, and Reduction in strength.

Another Way to Organize Them

- Local Optimizations
  - Constant Folding
  - Constant Propagation
  - Copy Propagation
  - Reduction in Strength
  - In Lining
  - Common sub-expression elimination
- Loop Optimizations
  - Loop Invariants
  - Reduction in strength due to induction variables
  - Loop unrolling
- Global Optimizations
  - Dead Code elimination
  - Code motion
  - Code hoisting
  - Register allocation
  - Interprocedural

Unnecessary Operations

- Common subexpression elimination
  - Save partial values instead of recomputing them
  - E.g. $a = (a^2 / (a^2 + b^2))$
- Dead code elimination
  - Don’t generate unreachable code
- Jump optimization
  - Avoid jumping to jump statements
- Useless code
  - Store X; Load X
Costly Operations: Constant Folding

- Don’t generate operations on constants; do them at compile time
  – Circ = 2 * PI * r
- If a variable will act like a constant for part or all of a program, transform its expressions also (constant propagation)

Constant Folding

- Subexpressions whose operands are all constants can be carried out at compile-time.
- E.g.:
  
  ```
  X := 2 * 4
  Rather than generating
  Mov 2 r5
  Mov 4 r7
  Prim '*' [r5,r7] r9
  Generate this instead
  Mov 8 r9
  . . .
  ```
- Code like this is not ordinarily written by programmers but is often the result of translation of index calculations.

Constant Propagation

- Sometimes we know a symbolic value is a constant. So we can propagate the constant and generate better code:
  
  ```
  1 step := 4
  2 total := 0
  3 i := 1
  4 total := x[i + step]
  ```
- Note that because there are no possible paths to 4 that do not pass through 1, 2 and 3 we know that i + step can be computed by (1+4) which is known at compile time to be 5.

Copy Propagation

- Assignments of one variable to another also propagate information:
  
  ```
  x := y
  . . .
  total := Z[x]
  ```
- Note if my translation knows that y is stored in some register, R7, I can use R7 rather than fetching x from memory
- Copy propagation my remove all references to x completely. This allow makes the assignment to x dead code and a candidate to further optimization.

Costly Operations: Reduction in Strength

- Don’t call a power function if you can multiply
  – (x*x vs x^2)
- Don’t multiply when you can shift
  – (x<<4 vs. 16*x)

Reduction in Strength

- Some sequences of code can be replaced with simpler (or less expensive) sequences.
  
  ```
  x := 2 * y
  could be replaced by
  x := y + y
  ```
- Exponentiation by 2 by multiply
  – x^2 == x * x
- Multiplication by factor of 2 by shift
Costly Operations: Function Calls

• Function calls have overhead
  – Create a new activation frame
  – Set up parameters & return value
  – Remove activation frame & jump to return
• Avoid by inlining (insert function code directly)
• Eliminate tail recursion with loop

Compiler Code Optimizations

• Inlining small functions
  – Repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations.
  – Inlining large functions will make the executable too large.

Inlining

• Some calls to functions (especially primitives like +, −, *, absolute value, ord and char) can be inlined as a sequence of machine instructions instead of a call to a library routine.

\[
i := \text{abs}(j)\\
Bneg \ j \ L2\\
Mov \ j \ i\\
Br \ L3\\
L2:\\\
Neg \ j \ R2\\
Mov \ R2 \ i\\
L3:\
\]

Common Sub-expressions

• Common subexpressions can be exploited by not duplicating code
  \[x := z[j+2] - w[j+2]\]
  \[T1 := j+2\]
  \[x := z[T1] - w[T1]\]

Common Sub-expressions

• Note that common subexpressions often occur even when they are not in the user level code.
  – E.g. Subscript computations on two multi-dimensional arrays with the same dimensions will often have common sub expressions even if the index to the arrays are completely different

Loop Invariants

• Computations inside loops which remain invariant each time around the loop can be computed once outside the loop rather than each time around the loop.
• Note that index calculation may also introduces computations which are invariant around the loop.
Loop Invariants

For $i := 1$ to $N$ do
  \{ total := x[i] / \text{sqr}(n) + total \}

$T_1 := \text{sqr}(n)$
For $i := 1$ to $N$ do
  \{ total := x[i] / T_1 + total \}

Induction Variables and reduction in strength

- Variables which vary in a regular way around loops are called induction variables.
- For loop variables are obvious cases, but implicit induction variables give much opportunity for optimization.

For $i := 1$ to 10 do
  \{ $k := i \times 4$; total := $x[i] + w[k]$ \}

Optimization

Induction Variables (2)

- Note that $k$ varies as a linear function of $i$.

\begin{verbatim}
i := 1
k := 4
while i <= 10 do
  \{ total := x[i] + w[k]
  i := i + 1; k := k + 4
  \}
\end{verbatim}

Optimization

Loop Unrolling

- Loop with low trip count can be unrolled. This does away with the loop initialization and test for termination conditions.

\begin{verbatim}
list := [1,2]
while (list <> nil) do
  \{ total := total + \text{hd}(list);
  i := i + 1; k := k + 4
  \}
\end{verbatim}

Dead Code Elimination

- Automatic generation techniques often generate code that is unreachable.

\begin{verbatim}
debuge := false;
if debug
  then print x;
f(x);
\end{verbatim}

- Because of constant propagation it is possible to tell at compile-time that the then branch will never be executed.

Compiler Code Optimizations

- Dead store elimination
  - If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.
Code hoisting

- Code hoisting
  - Moving computations outside loops
  - Saves computing time

Optimization

Code Hoisting (2)

- Code hoisting
  - In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.
    
    ```
    DO I = 1, 100
      ARRAY(I) = 2.0 * PI * I
    ENDDO
    
    - By introducing a temporary variable 't' it can be transformed to:
      
      ```
      t = 2.0 * PI
      DO I = 1, 100
        ARRAY(I) = t * I
      END DO
    ```

Optimization

Code Motion (reordering)

- Sometimes reordering statements that do not interfere, allows other more powerful optimizations to be come applicable.

```
Push R2
Mov R7 R3
Pop R4

Mov R7 R3
Push R2
Pop R4
Mov R7 R3
Mov R2 R4
```

- Now copy propagation might remove R2 altogether

Optimization

Code Motion (Code Hoisting)

- Branches in code sometimes repeat identical calculations.
- These calculations can sometimes be "hoisted" before the branch, then they don't have to be repeated.
- This saves space, but not time.

```java
if g(x)
  then  x := (d*2) + w / k
else  x := (d*2) - w / j

T1 := (d*2);
if g(x)
  then  x := T + w / k
else  x := T - w / j
```

- Multi branch "case" statements can make this quite a space saver

Optimization

Register Allocation

- Task: Manage scarce resources (registers) in environment with imperfect information (static program text) about dynamic program behavior.
- General aim is to keep frequently-used values in registers as much as possible, to lower memory traffic. Can have a large effect on program performance.
- Variety of approaches are possible, differing in sophistication and in scope of analysis used.

Optimization

Spilling

- Allocator may be unable to keep every "live" variable in registers; must then "spill" variables to memory. Spilling adds new instructions, which often affects the allocation analysis, requiring a new iteration.
- If spilling is necessary, what should we spill? Some heuristics:
  - Don't spill variables used in inner loops.
  - Spill variables not used again for "longest" time.
  - Spill variables which haven't been updated since last read from memory.
Simplistic approach

- Assume variables "normally" live in memory.
- Use existing (often redundant) fetches and stores present in IR1.
- So: only need to allocate registers to IR temporaries (T5 etc.).
- Ignore possibility of spills.
- Use simple linear scan register allocator based on liveness intervals.

Liveness

- To determine how long to keep a given variable (or temporary) in a register, need to know the range of instructions for which the variable is live.
- A variable or temporary is live immediately following an instruction if its current value will be needed in the future (i.e., it will be used again, and it won't be changed before that use).

Example

<table>
<thead>
<tr>
<th>live after instruction:</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2 := 3</td>
</tr>
<tr>
<td>T3 := T2</td>
</tr>
<tr>
<td>T4 := T3 + 4</td>
</tr>
<tr>
<td>T4 := T2 + T4</td>
</tr>
<tr>
<td>a := T4</td>
</tr>
<tr>
<td>(nothing)</td>
</tr>
</tbody>
</table>