Code Cleanup & Instruction Selection

March 23, 2010
Code Cleanup

Target is already close to assembly code

However, some features are inconvenient
  Two-way conditional jumps
  Statements and calls (side-effects) within expressions
  Nested function calls
Appel: a piece of Target code is said to be *canonical* if

Does not contain **SEQ** or **ESEQ**.

**CALL** appears only directly inside **EXP( . . . )** or **MOVE(TEMP t, . . .)**.

We can turn every **stm** into a list of …

We can turn every **exp** into a list of … and a …
ESEQ(EXP(CALL(NAME f,
    [CONST 1,
    BINOP(MINUS,
    CALL(NAME g, [CONST 2]),
    CONST 3)]),
    BINOP(PLUS,
    ESEQ(SEQ(ASSIGN(TEMP t, CONST 4),
    ASSIGN(TEMP u, CONST 5)),
    CONST 6),
    ESEQ(ASSIGN(TEMP w, CONST 7),
    CALL(NAME h, [])))))
Basic Blocks

We now have a linear list of statements.

Break these up into *basic blocks* *ala* Appel

- First statement is `LABEL`
- Last statement is `JUMP` or `CJUMP`
- No other `LABEL` or `JUMP/CJUMP`
Algorithm: Scan statements sequentially

Start new block at every **LABEL**

End block at every **JUMP** or **CJUMP**

Insert a fresh label next if needed

Add a **JUMP** to done at the end if needed
Given Appel's definition of basic blocks, the blocks themselves can be arranged in any order.

What would a good order be?
Trace Generation

Simple algorithm:

- Keep track of which nodes have been made part of a trace.
- Follow a path through the graph, traversing only nodes that are not already part of a trace.
  - Similar to DFS
  - Stop when you get stuck
- Repeat until all blocks are part of a trace
Trace Optimizations

Some compilers are pickier about the traces they generate
What would properties of a good trace be?

Could traces overlap?

Particularly common for VLIW or highly superscalar machines.
Trace Cleanup

The program is now a sequence of traces
List of (basic block lists)
Equivalent to a list of basic blocks
Walk through all the blocks
If it ends in a CJUMP and is followed by the false label, do nothing
If it ends in a CJUMP and is followed by the true label, reverse the CJUMP
If it ends in a CJUMP and is followed by neither label, follow it with a new false block that jumps to the old false label.
If it ends in a JUMP followed by its destination, delete the jump.
Now that intermediate code is in a nice form, how do we generate machine instructions?

Idea: tree patterns.

Algorithms:
- Maximal Munch
- Dynamic Programming
Maximal Munch

Greedy Algorithm.

- Always pick the largest possible tile at the root of the tree
- Then recursively tile the remaining subtrees
- Always succeeds under reasonable assumptions
  - e.g., if there is a single-node tile for every sort of node
- Results in tiling that is optimal but not the optimum.
Finds optimum tiling, not just optimal

Idea:
- Examine all possible tiles at the root
- See what the resulting cost of the tree will be assuming optimum tiling of the remaining subtrees
- Use dynamic programming (memoizing) to avoid recomputing the optimal tiling of a subtree.

Two-pass algorithm:
- First find optimum tiling of every subtree
- Then emit the code for the optimum tiling of entire tree.
Dynamic Programming Issues

- Can be messy to program efficiently
  - Particularly for non-RISC machines
  - There exist automatic tools for generating code generators
  - Input looks like a context-free grammar
- Optimum only under unrealistic cost model
  - Assumes the cost of a tiling is the sum of the costs of the individual tiles.
Handling CISC Machines

Hard to generate good code for CISC machines. Why?
Appel's Suggestions

Depend heavily on the register allocator

Few registers? Generate code assuming infinite number of registers, just as for RISC machines

Instruction-specific registers and/or two-address instructions? Insert extra moves.

\[ t_1 \leftarrow t_2 \times t_3 \] becomes

\[
\begin{align*}
\text{mov eax, t2} & \quad (\text{eax} \leftarrow t2) \\
\text{mul t3} & \quad (\text{eax} \leftarrow \text{eax} \times t3) \\
\text{mov t1, eax} & \quad (t1 \leftarrow \text{eax})
\end{align*}
\]

Hope that register allocator eliminates moves.

Ignore autoincrement, fancy addressing modes