CMPT 755
Compilers
Anoop Sarkar
http://www.cs.sfu.ca/~anoop
Code Generation

- Instruction selection
- Register allocation
- Stack frame allocation
- Static or global allocation
- Basic blocks and Flow graphs
- Transformations on Basic blocks
Code Generation

- Produce code that is correct
- Produce code that is of high quality (size and speed)
- The problem of generating optimal code is undecidable
- In practice, we need heuristics that generate good, but perhaps not optimal, code
Instruction Costs

• Since optimal code generation is not possible a useful way to think about the problem is as an optimization problem
• Each instruction can be assigned a cost
  – For complex instruction sets some instructions can be more preferable than others
• Using registers have zero cost, while using memory locations is costlier
• If each instruction is equally expensive, this will minimize the number of instructions as well
Register Allocation

- Code generation either directly to assembly or from 3-address code (TAC)
- For each location, we have to find a register to store values or temporary values
  - Problem: limited number of registers
- Compiler has to find optimal assignment of locations to registers
  - Register use can involve stacked temporaries or other ways to reuse registers
- If no more registers available, we *spill* a location into memory
Register Allocation

• Bind locations to registers for all or part of a function

• Dynamic Optimization Problem
  – Not compile-time, but run-time frequency is what counts

• Heuristics
  – Allocate registers for variables likely to be used frequently
  – Keep temporaries in registers $\rightarrow$ minimize their number
Basic Blocks

- Functions transfer control from one place (the caller) to another (the called function).
- Other examples include any place where there are branch instructions.
- A basic block is a sequence of statements that enters at the start and ends with a branch at the end.
- Remaining task of code generation is to create code for basic blocks and branch them together.
Blocks

main()
{
    int a = 0; int b = 0;
    {
        int b = 1;
        {
            int a = 2; printf("%d %d\n", a, b);
        }
        {
            int b = 3; printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
Partition into Basic Blocks

- Input: sequence of TAC instructions
  1. Determine set of leaders, the 1st statement of each basic block
     a) The 1st statement is a leader
     b) Any statement that is the target of a conditional jump or goto is a leader
     c) Any statement immediately following a conditional jump or goto is a leader
  2. For each leader, the basic block contains all statements upto the next leader
Control Flow Graph (CFG)

```c
int main() {
    extern int f(int);
    int i;
    int *a;
    for (i = 0;
        i < 10;
        i = i + 1)
    {
        a[i] = f(i);
    }
}
```
main:
    BeginFunc 72 ;
    i = 0 ;
_L0:
    _tmp1 = 10 ;
    _tmp2 = i < _tmp1 ;
    IfZ _tmp2 Goto _L1 ;
    _tmp3 = 4 ;
    _tmp4 = _tmp3 * i ;
    _tmp5 = a + _tmp4 ;
    param i #0 ;
    _tmp6 = call f ;
    pop 4 ;
    *((_tmp5) = _tmp6 ;
    _tmp7 = 1 ;
    i = i + _tmp7 ;
    goto _L0 ;
_L1:
    EndFunc ;
Dataflow Analysis

• Compute Dataflow Equations over Control Flow Graph
  – Reaching Definitions (Forward)
    \[ \text{out}[BB] := \text{gen}[BB] \cup (\text{in}[BB] - \text{kill}[BB]) \]
    \[ \text{in}[BB] := \bigcup \text{out}[s] : \forall s \in \text{pred}[BB] \]
  – Liveness Analysis (Backward)
    \[ \text{in}[BB] := \text{use}[BB] \cup (\text{out}[BB] - \text{def}[BB]) \]
    \[ \text{out}[BB] := \bigcup \text{in}[s] : \forall s \in \text{succ}[BB] \]
• Computation by fixed-point analysis
Liveness Analysis

Entry

IN = {} → i = 0 → OUT = {i} → IN = {a, i}

_L0:
$tmp1 = 10 ;$
$tmp2 = i < _tmp1 ;$
ifz $tmp2$ goto _L1 ;

$tmp3 = 4 ;$
$tmp4 = _tmp3 * i ;$
$tmp5 = a + _tmp4 ;$
param i #0 ;
$tmp6 = call f ;$
pop 4 ;
$(_tmp5) = _tmp6 ;$
	$(._tmp5) = _tmp6 ;$
$tmp7 = 1 ;$
i = i + _tmp7 ;$
goto _L0 ;$

OUT = {i} → IN = {a, i}

OUT = {a, i}

IN = {a, i} → OUT = {a, i}

IN = {a, i}

IN = {a, i}

IN = {} → Exit
Register Allocation

• Do liveness analysis on Control Flow Graph
  – Straightforward (iteration-less) computation within basic block
  – Compute live ranges for each location

• Build interference graph
  – Two locations are connected if their live ranges overlap
Interference Graph
Colored Interference Graph
Register Allocation as Graph Coloring

- First pass: use as many symbolic registers as needed including registers for stack pointers, frame pointers, etc.

- Register Interference Graph
  - Two nodes in the graph are connected if their live ranges overlap

- Color interference graph
  - Result is register assignment -- $k$ colors for $k$ registers
Register Allocation as Graph Coloring

• Second pass: assign physical registers to symbolic ones
  – Construct a register interference graph
    (nodes are symbolic registers and edge denotes that they cannot be assigned to the same physical register)
  – Attempt to $k$-color the interference graph, where $k$ is the number of available registers
  – $k$-coloring a graph is NP-complete
Register Allocation as Graph Coloring

• Algorithm for solving whether a graph $G$ is $k$-colorable:
  • Pick any node $n$ with fewer than $k$ neighbours
  • Remove $n$ and adjacent edges to create a new graph $G'$
  • $k$-coloring of $G'$ can be extended to $k$-coloring to $G$ by assigning to $n$ a color that is not assigned to any of $n$’s neighbours
  • If we cannot extend $G'$ to $G$, then $k$-coloring of $G$ is not possible
Register Allocation as Graph Coloring

- If every node in G has more than $k$ neighbours, $k$-coloring of G is not possible.
- Take some node $n$ and spill into memory, remove it from the graph and continue $k$-coloring.
- Spilling = generating code to store contents of register to memory and when location is used generate code to load from memory into an available register (by spilling another location).
Register Allocation as Graph Coloring

- Many different heuristics for picking a node $n$ to spill
- E.g. avoid introducing spilling symbolic registers that are inside loops or heavily visited regions of code
- C allows a `register` and a `volatile` keyword to direct the compiler whether a variable contains a value that is heavily used.
- Special case: Register Allocation for Expression Trees (Maximal Munch suffices for this task)
Summary

- Code generation: from Intermediate Representation (IR) to Assembly
- Three Address Code (TAC) can be easily converted to a control flow graph
- The control flow graph allows sophisticated dataflow analysis
- The liveness of each location can be used for register allocation
- Register Allocation as heuristic graph coloring.