Intro to optimizations

- target code generated from internal/intermediate representation for the program in question is rarely in optimal form
- opportunities for “improvement” creep into the IR in many ways
- original source code may have contained inefficiencies
- initial generic transformations from HLL source code statements to the target language doesn't account for wider context, chances to improve based on things “outside” the immediate statement
- target language instructions might provide specific pitfalls and/or opportunities for optimization not available in the source

Goals of optimization

- usually execution speed is a priority
- other possible goals may include
  - size of memory used
  - number of memory accessed made
  - size of executable produced
  - response time to external events
  - time required to compile
- often trade-off poorer performance in one area to get better performance in another

Opportunities for optimization

- We've seen some, there are many more, e.g.:
  - eliminating redundant computations
  - eliminating redundant loads from memory
  - taking advantage of memory heirarchy
  - inlining functions
  - unrolling, splitting, inverting, interchanging loops
  - eliminating unreachable code, unused variables
  - reducing jump counts
  - parallelizing
  - adding subroutines to eliminate redundancy

Safety: equivalent behaviour

- any optimization involves changing the code
- want to ensure that the changes still produce equivalent behaviour
  - will treat two expressions as equivalent if, *in the context of the program*, they produce identical results
- aside: the transformations generally make it more difficult for the programmer to see the relationship between the original source code and the target language code produced

Scope of optimizations

- can apply optimizations to different “layers” of code:
  - local: (sequential) block level
  - regional: multiple blocks e.g. Loops, if/else structures
  - intraprocedural: subroutine level (aka global)
  - interprocedural: whole program
- later in the process will also consider:
  - peephole: tiny window of several generated instructions

Local optimizations

- a single block of sequential statements (single entry point, single exit point)
- easier to analyze because we know every instruction is always run, and run in a fixed order
- some common opportunities:
  - eliminating redundant operations
  - constant folding
  - selection of naming schemes
  - balancing tree heights for parallelization

Regional optimizations

- span more than a blocks, e.g. code for a loop, switch, etc
- compiler needs to select a region (extending blocks)
- many loop optimizations take place here
  - interchange (swap inner/outer nested loops)
  - inversion (switching top/bottom tested)
  - unswitching (swap inner/outer if statement and loop)
  - splitting (partition big loop into multiple smaller ones)
  - unrolling (replace loop with sequence of statements)
  - moving invariants out of the loop body

Intraprocedural optimizations

- cover a whole subroutine (a.k.a. global, to be confusing)
- reveals coordination of data across blocks/regions
- some form of data flow analysis needed
- optimizations might include
  - eliminating uninitialized variables
  - eliminating unused variables
  - eliminating dead code
  - taking advantage of asymmetric branch costs

Interprocedural optimizations

- whole-program optimizations
- get the whole picture, but using less specific detail about the inner workings (somewhat top-down view)
- call graphs and dataflow analysis again
- optimizations might include
  - inline expansions
  - revised handling of caller/callee responsibilities
  - subroutine placement (localize funcs that call each other)

Code gen optimizations

- later considerations, once the code generation is nearly complete, may revisit last-stage optimizations
  - instruction selection and scheduling
  - peephole optimizations
  - register allocation
  - recalculating values to avoid memory loads