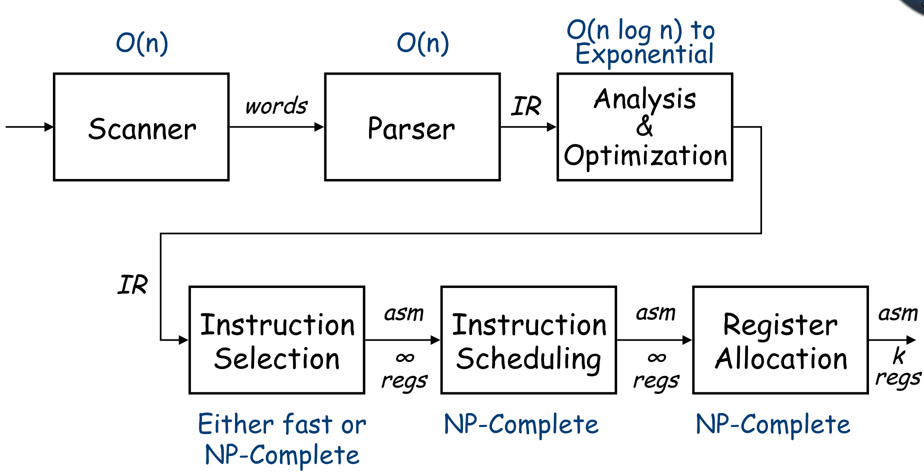
COMP 412
FALL 2009

Introduction to Code Generation

Comp 412

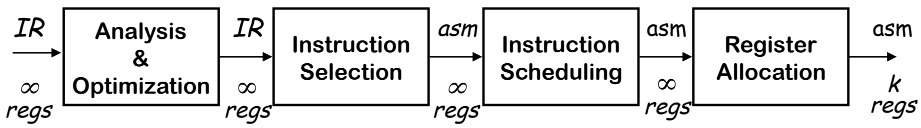
Structure of a Compiler



A compiler is a lot of fast stuff followed by some hard problems
- The hard stuff is mostly in code generation and optimization
- For superscalars, it is allocation & scheduling that count

Structure of a Compiler

For the rest of 412, we assume the following model



- Selection is fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- Operation placement is not yet critical *(unified register set)*

What about the IR ?

- Low-level, RISC-like IR such as ILOC
- Has "enough" registers
- ILOC was designed for this stuff

{ Branches, compares, & labels
Memory tags
Hierarchy of loads & stores
Provision for multiple ops/cycle

Definitions

Instruction selection

- Mapping <u>*IR*</u> into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

These 3 problems are tightly coupled.

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

The Big Picture

How hard are these problems?

Instruction selection
- Can make locally optimal choices, with automated tool
- Global optimality is (undoubtedly) NP-Complete

Instruction scheduling
- Single basic block $\Rightarrow$ heuristics work quickly
- General problem, with control flow $\Rightarrow$ NP-Complete

Register allocation
- Single basic block, no spilling, & 1 register size $\Rightarrow$ linear time
- Whole procedure is NP-Complete

The Big Picture

Conventional wisdom says that we lose little by solving these problems independently

Instruction selection
- Use some form of pattern matching
- Assume enough registers or target "important" values

Instruction scheduling
- Within a block, list scheduling is "close" to optimal
- Across blocks, build framework to apply list scheduling

Optimal for > 85% of blocks

Register allocation
- Start from virtual registers & map "enough" into *k*

This slide is full of "fuzzy" terms

The Big Picture

What are today's hard issues issues?

Instruction selection

- Making actual use of the tools
- Impact of choices on power and on functional unit placement

Instruction scheduling

- Modulo scheduling loops with control flow
- Schemes for scheduling long latency memory operations
- Finding enough ILP to keep functional units (& cores) busy

Register allocation

- Cost of allocation, particularly for JITs
- Better spilling (*space & speed*)?
- Meaning of optimality in SSA-based allocation

Code Shape - the Next Chapter

Definition

- All those nebulous properties of the code that effect performance
- Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations
- Code shape is the end product of many decisions *(big & small)*

Impact

- Code shape influences algorithm choice & results
- Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible

- Example: explicit branch targets in ILOC simplify analysis
- Example: hierarchy of memory operations in ILOC *(EaC, p 237)*

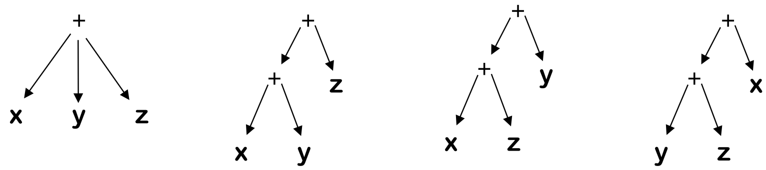
See Morgan's book for more ILOC examples

Code Shape

Addition is commutative & associative for integers

My favorite example

x + y + z	x + y → t1	x + z → t1	y + z → t1
	t1+ z → t2	t1+ y → t2	t1+ z → t2



- What if x is 2 and z is 3?
- What if y+z is evaluated earlier?

The "best" shape for x+y+z depends on contextual knowledge

— There may be several conflicting options

Code Shape

Another example -- the case statement

- Implement it as cascaded if-then-else statements
  — Cost depends on where your case actually occurs
  — O(number of cases)
- Implement it as a binary search
  — Need a dense set of conditions to search
  — Uniform (log n) cost
- Implement it as a jump table
  — Lookup address in a table & jump to it
  — Uniform (constant) cost

Performance depends on order of cases!

Compiler must choose best implementation strategy

No amount of massaging or transforming will convert one into another

Code Shape

Why worry about code shape? Can't we just trust the optimizer and the back end?

- Optimizer and back end approximate answers to many hard problems
- The compiler's individual passes must run quickly
- It often pays to encode useful information into the IR
  - Shape of an expression or a control structure
  - A value kept in a register rather than in memory
- Deriving such information would be expensive
- Writing it down in the IR is often easier and cheaper