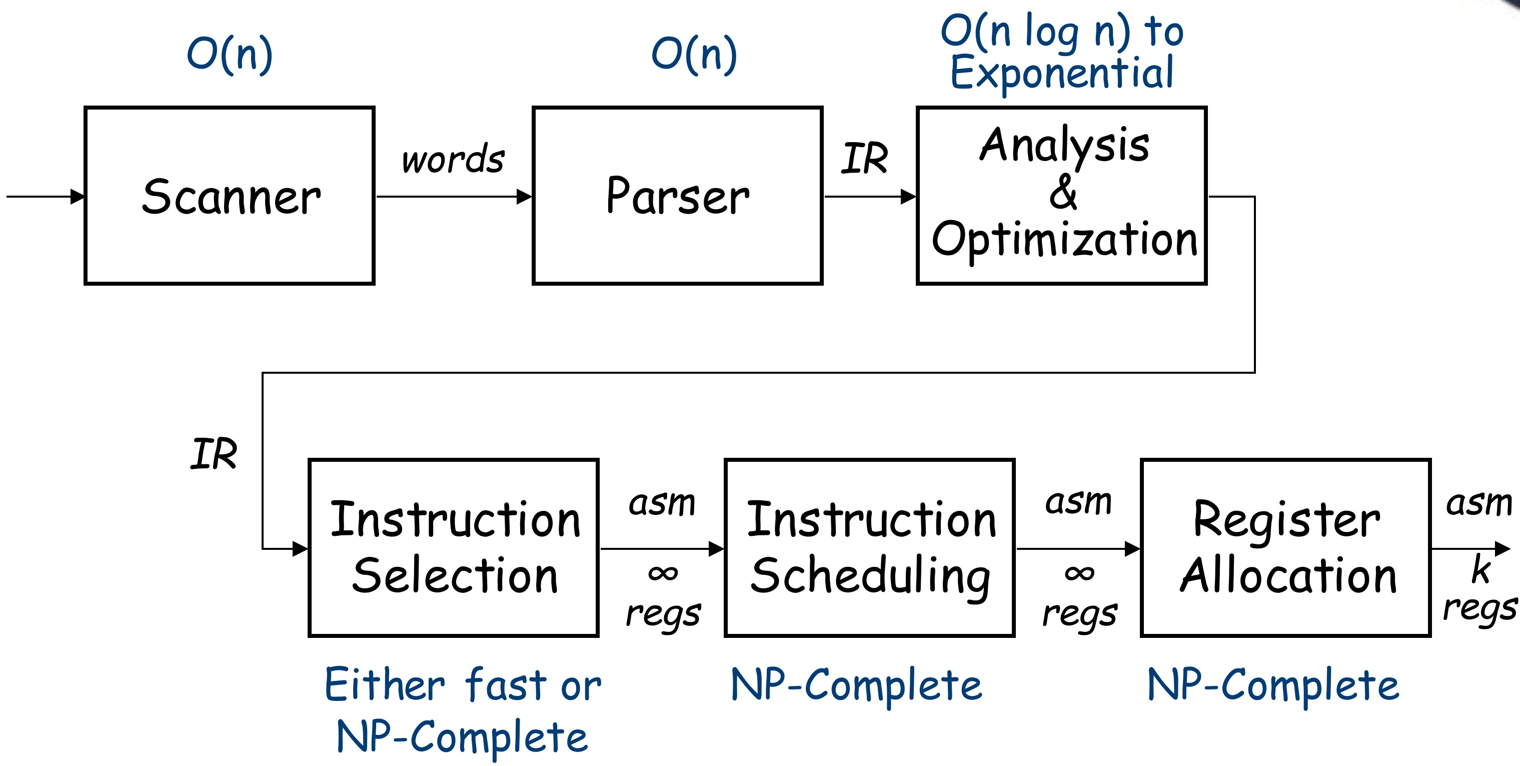
COMP 412
FALL 2010

Introduction to Code Generation

Comp 412

Structure of a Compiler

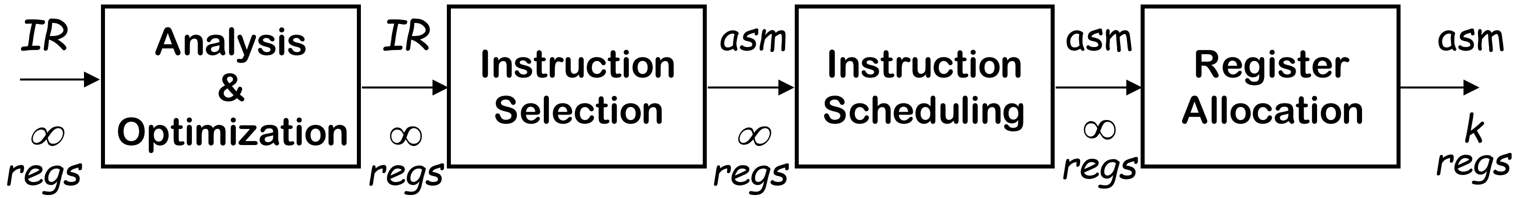


A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in code generation and optimization
- For multicores, we need to manage parallelism & sharing
- For unicore performance, allocation & scheduling are critical

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

Parallelism either comes from the application, or is derived with techniques that are beyond the level of this course.

Definitions

Instruction selection

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

These 3 problems are tightly coupled.

Instruction scheduling

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

Register allocation

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

Code Shape (Chapter 7)

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

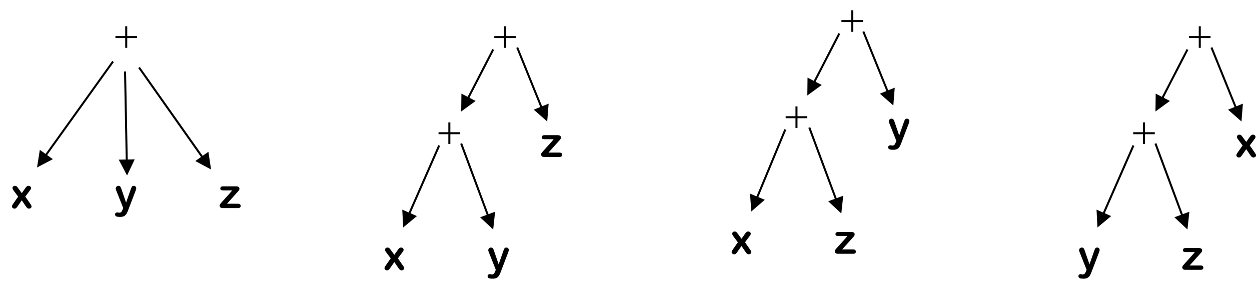
- Example: explicit branch targets in ILOC simplify analysis
- Example: hierarchy of memory operations in ILOC *(EaC, p 237)*

See Morgan's book for more ILOC examples

Code Shape

My favorite example

x + y + z	x + y $\rightarrow$ t1	x + z $\rightarrow$ t1	y + z $\rightarrow$ t1
	t1+ z $\rightarrow$ t2	t1+ y $\rightarrow$ t2	t1+ z $\rightarrow$ 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

Addition is commutative & associative for integers

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 the 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 may be expensive, when possible
- Recording it explicitly in the IR is often easier and cheaper

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

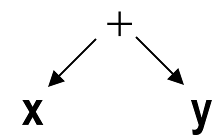
The Concept

- Assume an AST as input & ILOC as output
- Use a postorder treewalk evaluator (*visitor pattern* in OOD)
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Bury complexity of addressing names in routines that it calls
  - *base*(), *offset*(), & *val*()
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
  }
  return result;
}
```

Example:



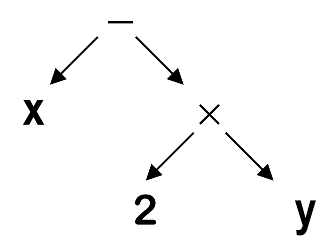
Produces:

expr("x") →		
loadI	@x	⇒ r1
loadAO	r_{arp},r1	⇒ r2
expr("y") →		
loadI	@y	⇒ r3
loadAO	r_{arp},r3	⇒ r4
NextRegister() → r5		
Emit(add,r2,r4,r5) →		
add	r2,r4	⇒ r5

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,– :
        t1← expr(left child(node));
        t2← expr(right child(node));
        result ← NextRegister();
        emit (op(node), t1, t2, result);
        break;
     case IDENTIFIER:
        t1← base(node);
        t2← offset(node);
        result ← NextRegister();
        emit (loadAO, t1, t2, result);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadI, val(node), none, result);
        break;
   }
   return result;
}
```

Example:



Produces:

loadI	@x	⇒ r1
loadAO	r_{arp},r1	⇒ r2
loadI	2	⇒ r3
loadI	@y	⇒ r4
loadAO	r_{arp},r4	⇒ r5
mult	r3,r5	⇒ r6
sub	R2,r6	⇒ r7

Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values that reside in registers?
  - Modify the IDENTIFIER case
  - Already in a register $\Rightarrow$ return the register name
  - Not in a register $\Rightarrow$ load it as before, but record the fact
  - Choose names to avoid creating false dependences
- What about parameter values?
  - Many linkages pass the first several values in registers
  - Call-by-value $\Rightarrow$ just a local variable with a negative offset
  - Call-by-reference $\Rightarrow$ negative offset, extra indirection
- What about function calls in expressions?
  - Generate the calling sequence & load the return value
  - Severely limits compiler's ability to reorder operations

Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables
  - Original PL/I had asymmetric tables for BCD & binary integers

Typical Table for Addition

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Extending the Simple Treewalk Algorithm

What about evaluation order?

- Can use commutativity & associativity to improve code
- This problem is truly hard

Local rather than global

Commuting operands at a single operation is much easier

- 1st operand must be preserved while 2nd is evaluated
- Takes an extra register for 2nd operand
- Should evaluate more demanding operand expression first

(Ershov in the 1950's, Sethi in the 1970's)

Taken to its logical conclusion, this creates Sethi-Ullman scheme for register allocation [See Bibliography in EaC]

Generating Code in the Parser

Need to generate an initial IR form

- Chapter 4 talks about ASTs & ILOC
- Might generate an AST, use it for some high-level, near-source work such as type checking and optimization, then traverse it and emit a lower-level IR similar to ILOC for further optimization and code generation

The Big Picture

- Recursive treewalk performs its work in a bottom-up order
  - Actions on non-leaves occur after actions on children
- Can encode same basic structure into *ad-hoc* SDT scheme
  - Identifiers load themselves & stack virtual register name
  - Operators emit appropriate code & stack resulting VR name
  - Assignment requires evaluation to an *lvalue* or an *rvalue*

Ad-hoc SDT versus a Recursive Treewalk

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
}
```

```
Goal :      Expr      { $$ = $1; } ;
Expr:       Expr PLUS Term
            { t = NextRegister();
              emit(add,$1,$3,t); $$ = t; }
       |    Expr MINUS Term     {…}
       |    Term     { $$ = $1; } ;
Term:       Term TIMES Factor
            { t = NextRegister();
              emit(mult,$1,$3,t); $$ = t; };
       |    Term DIVIDES Factor    {…}
       |    Factor { $$ = $1; };
Factor:     NUMBER
            { t = NextRegister();
              emit(loadI,val($1),none, t );
              $$ = t; }
       |    ID
             { t1 = base($1);
               t2 = offset($1);
               t = NextRegister();
              emit(loadAO,t1,t2,t);
              $$ =  t; }
```


Some details omitted for space; see Chapter 4.

Handling Assignment (just another operator)

lhs $\leftarrow$ ***rhs***

Strategy

- Evaluate *rhs* to a value *(an rvalue)*
- Evaluate *lhs* to a location *(an lvalue)*
  - *lvalue* is a register $\Rightarrow$ move rhs
  - *lvalue* is an address $\Rightarrow$ store rhs
- If *rvalue* & *lvalue* have different types
  - Evaluate *rvalue* to its "*natural*" type
  - Convert that value to the type of **lvalue*

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Keeping ambiguous values in memory lets the hardware sort out the addresses.

Handling Assignment

What if the compiler cannot determine the type of the rhs?

- Issue is a property of the language & the specific program
- For type-safety, compiler must insert a run-time check
  - Some languages & implementations ignore safety (*bad idea*)
- Add a *tag* field to the data items to hold type information
  - Explicitly check tags at runtime

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
  then
    convert rhs to type(lhs) or
    signal a run-time error
lhs ← rhs
```

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time

Handling Assignment

Compile-time type-checking

- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

Can *design* the language so all checks are static

Handling Assignment with Reference Counts

Reference counting is an incremental strategy for implicit storage deallocation (*alternative to batch collectors*)

- Simple idea
  - Associate a count with each heap allocated object
  - Increment count when pointer is duplicated
  - Decrement count when pointer is destroyed
  - Free when count goes to zero
- Advantages
  - Smaller collections amortized over all pointer assignments
    - → Useful in real-time applications, user interfaces
  - Counts will be in cache, ILP may reduce expense
- Disadvantages
  - Freeing root node of a graph implies a lot of work & disruption
    - → Can adopt a protocol to bound the work done at each point
  - Cyclic structures pose a problem

Handling Assignment with Reference Counts

Implementing reference counts

- Must adjust the count on each pointer assignment
- Extra code on every counted (*e.g.*, pointer) assignment

Code for assignment becomes

```
evaluate rhs
lhs→count--
lhs ← addr(rhs)
rhs→count++
if (rhs→count = 0)
    free rhs
```

Likely hits in the L1 cache

This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, the overhead may become either cheap or free …

Evaluating Expressions

What is left for next class?

- Boolean & relational values
  - Need to extend the grammar
  - Need to represent the value, explicitly or implicitly
- Loading values that are more complex than scalars
  - Array elements, structure elements, characters in a string

Extra Slides Start Here

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