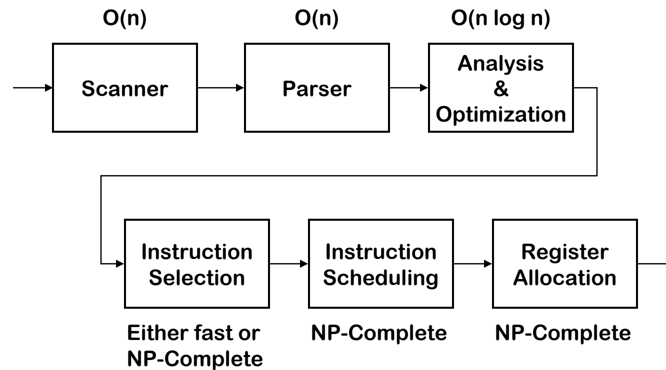
Structure of a Compiler

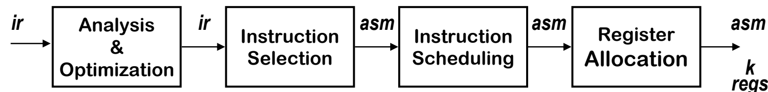


A compiler is a lot of fast stuff followed by some truly hard problems

- The hard stuff is mostly in code generation and optimization
- For superscalars, its allocation & scheduling that count

Structure of a Compiler

For the rest of 430, 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 called 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 *ir* into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

These 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

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 (*>85% of cases*)
- Across blocks, build framework to apply list scheduling

Register allocation

- Start from virtual registers & map "enough" into *k*
- With targeting, focus on good priority heuristic

This slide is full of "fuzzy" terms

Code Shape

Definition

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

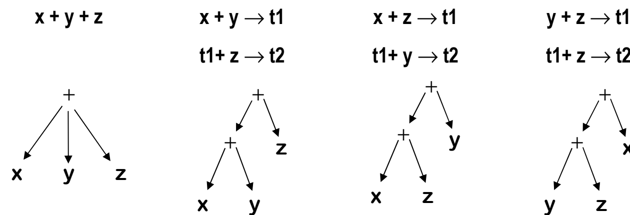
Impact

- 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: branch targets in ILOC are explicit to simplify analysis
- Example: hierarchy of memory operations in ILOC

Code Shape



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

Addition is commutative & associative for integers

The "best" shape for the 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

Compiler must choose best implementation strategy

No amount of massaging or transforming will convert one into another

Generating Code for Expressions

The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - > When only 1 name can reference its value
  - > Pointers, parameters, aggregates & arrays all cause trouble
- When should a value be allocated to a register?
  - > When it is both *safe* & *profitable*

Encoding this knowledge into the *IR*

- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference
- ILOC has textual "memory tags" on loads, stores, & calls
- ILOC has a hierarchy of loads & stores

Relies on a strong register allocator

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

- Use a simple treewalk evaluator
- Bury complexity in routines it calls
  - > *base*(), *offset*(), & *val*()
- Implements expected behavior
  - > Visits & evaluates children
  - > Emits code for the op itself
  - > Returns register with result
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

Code generation logic

As we will see in a few weeks, addressing of identifiers is a 2 step process:

- Base address is the start of the local stack frame (activation record, i.e., storage for a particular procedure)
- Offset is the location of that identifier within a given activation record

***base*()**

Makes sure the activation record pointer is in a register

offset()

Makes sure the offset of a given variable within an activation record is in a register

val()

makes sure that the argument value is in a register

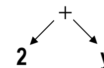
Assumption – for now – Assume only 1 procedure whose activation record address is in register r0.

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); // t2→nextregister()
        result ← NextRegister();
        emit (loadAO, t1, t2, result);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadI, val(node), none, result);
        break;
     }
     return result;
}
```

Example:

Node(+,2,y)



Produces:

expr("2") →

loadI 2 ⇒ r1

NextRegister() → r2

expr("y") →

loadI @y ⇒ r2

loadAO r0, r2 ⇒ r3

NextRegister() → r4

emit(add,r1,r3,r4) →

add r1, r3 ⇒ r4

Generating Code for Expressions

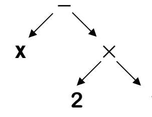
```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case ×,÷,+,– :
      t1← expr(left child(node));
      t2← expr(right child(node));
      result ← NextRegister();
      emit (op(node), t1, t2, result);
      break;
    case IDENTIFIER:
      t1← base(node);
      t2← offset(node);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

Example:

node(-,x,

node(×,2,y))

Generates:



loadI	@x	⇒ r1
loadAO	r0, r1	⇒ r2
loadI	2	⇒ r3
loadI	@y	⇒ r4
loadAO	r0,r4	⇒ r5
mult	r3, r5	⇒ r6
sub	r2, r6	⇒ r7

Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values in registers?
  - Modify the IDENTIFIER case
  - Already in a register ⇒ return the register name
  - Not in a register ⇒ 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 ⇒ just a local variable with “funny” offset
  - Call-by-reference ⇒ needs an 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

Typical Addition Table

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

Other code patterns – Boolean expressions

Assume each operator pops top argument from stack:

- E1 = E2
```
E1
E2
if_equal L1
iconst_0
goto L2
L1:
iconst_1
L2:
```
- E1<E2
```
E1
E2
if_lessthan L1
iconst_0
goto L2
L1:
iconst_1
L2:
```
- E1 && E2
```
E1
iconst_1
if_notequal L1
pop
E2
L1:
```
- E1||E2
```
E1
iconst_1
if_equal L1
pop
E2
L1:
```
- !E
```
E
iconst_1
if_equal L1
pop
iconst_1
Goto L2
L1:
pop
Iconst_0
L2:
```

Control structures

- X=E
```
x
E
Store
```
- If (E) S
```
E
iconst_0
if_equal L1
S
L1:
Pop
```
- If (E) S1 else S2
```
E
iconst_0
if_equal L1
S1
Goto L2
L1:
S2
L2:
Pop
```
- While (E) S
```
L2:
E
iconst_0
if_equal L1
S1
pop
Goto L2
L1:
pop
```


Extending the Simple Treewalk Algorithm

What about evaluation order?

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

What about order of evaluating operands?

- 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

Generating Code in the Parser

Need to generate an initial IR form

- Might want to generate a linear form, such as ILOC

The big picture

- Recursive algorithm really works bottom-up
  - > Actions on non-leaves occur after children are done
- 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*
    - → Some modal behavior will be necessary

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; }
```

What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent* *(and has been since the 60's)*

Assumed latencies *(conservative)*

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - > Non-blocking ⇒fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - > Fill slots with unrelated operations
  - > Percolates branch upward
- Scheduler should hide the latencies

Example

$w \leftarrow w * 2 * x * y * z$

Simple schedule

1	loadAI	r0,@w	⇒ r1
4	add	r1,r1	⇒ r1
5	loadAI	r0,@x	⇒ r2
8	mult	r1,r2	⇒ r1
9	loadAI	r0,@y	⇒ r2
12	mult	r1,r2	⇒ r1
13	loadAI	r0,@z	⇒ r2
16	mult	r1,r2	⇒ r1
18	storeAI	r1	⇒ r0,@w
21	r1 is free		

2 registers, 20 cycles

Schedule loads early

1	loadAI	r0,@w	⇒ r1
2	loadAI	r0,@x	⇒ r2
3	loadAI	r0,@y	⇒ r3
4	add	r1,r1	⇒ r1
5	mult	r1,r2	⇒ r1
6	loadAI	r0,@z	⇒ r2
7	mult	r1,r3	⇒ r1
9	mult	r1,r2	⇒ r1
11	storeAI	r1	⇒ r0,@w
14	r1 is free		

3 registers, 13 cycles

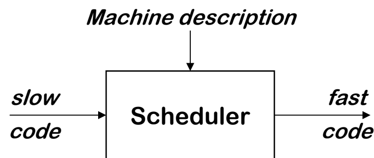
Reordering operations for speed is called instruction scheduling

Instruction Scheduling *(Engineer's View)*

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

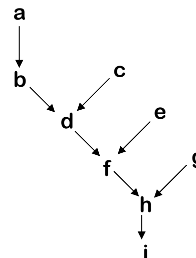
Instruction Scheduling *(The Abstract View)*

To capture properties of the code, build a <u>precedence graph</u> *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ if & only if n_2 uses the result of n_1

a:	loadAI	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAI	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAI	r0,@y	⇒ r2
f:	mult	r1,r2	⇒ r1
g:	loadAI	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAI	r1	⇒ r0,@w

The Code



The Precedence Graph

Instruction Scheduling *(Definitions)*

A correct schedule S maps each $n \in N$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1, n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue

The length of a schedule S, denoted $L(S)$, is

$L(S) = max_{n \in N} (S(n) + delay(n))$

The goal is to find the shortest possible correct schedule.

S is time-optimal if $L(S) \leq L(S_1)$, for all other schedules S_1

A schedule might also be optimal in terms of registers, power, or space….

Instruction Scheduling *(What's so difficult?)*

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simple case

- Restricted to straight-line code
- Consistent and predictable latencies

Instruction Scheduling

The big picture

1. Build a precedence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

Local List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
  if (Ready ≠ Ø) then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op

  Cycle ← Cycle + 1

  for each op ∈ Active
    if (S(op) + delay(op) ≤ Cycle) then
      remove op from Active
      for each successor s of op in P
        if (s is ready) then
          Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, put it on Ready

Detailed Scheduling Algorithm I

Idea: **Keep a collection of worklists W[c], one per cycle**

> **We need MaxC = max delay + 1 such worklists**

Code:

```
for each n ∈ N do begin count[n] := 0; earliest[n] = 0 end
for each (n1,n2) ∈ E do begin
    count[n2] := count[n2] + 1;
    successors[n1] := successors[n1] ∪ {n2};
end
for i := 0 to MaxC – 1 do W[i] := ∅;
Wcount := 0;
for each n ∈ N do
    if count[n] = 0 then begin
        W[0] := W[0] ∪ {n}; Wcount := Wcount + 1;
    end
c := 0;  // c is the cycle number
cW := 0;// cW is the number of the worklist for cycle c
instr[c] := ∅;
```


Detailed Scheduling Algorithm II

```
while Wcount > 0 do begin
    while W[cW] = ∅ do begin
        c := c + 1; instr[c] := ∅; cW := mod(cW+1,MaxC);
    end
    nextc := mod(c+1,MaxC);
    while W[cW] ≠ ∅ do begin
Priority ——→ select and remove an arbitrary instruction x from W[cW];
        if ∃ free issue units of type(x) on cycle c then begin
            instr[c] := instr[c] ∪ {x}; Wcount := Wcount - 1;
            for each y ∈ successors[x] do begin
                count[y] := count[y] – 1;
                earliest[y] := max(earliest[y], c+delay(x));
                if count[y] = 0 then begin
                    loc := mod(earliest[y],MaxC);
                    W[loc] := W[loc] ∪ {y}; Wcount := Wcount + 1;
                end
            end
        else W[nextc] := W[nextc] ∪ {x};
    end
end
```

More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready $\Rightarrow$ all operands available	• Ready $\Rightarrow$ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors