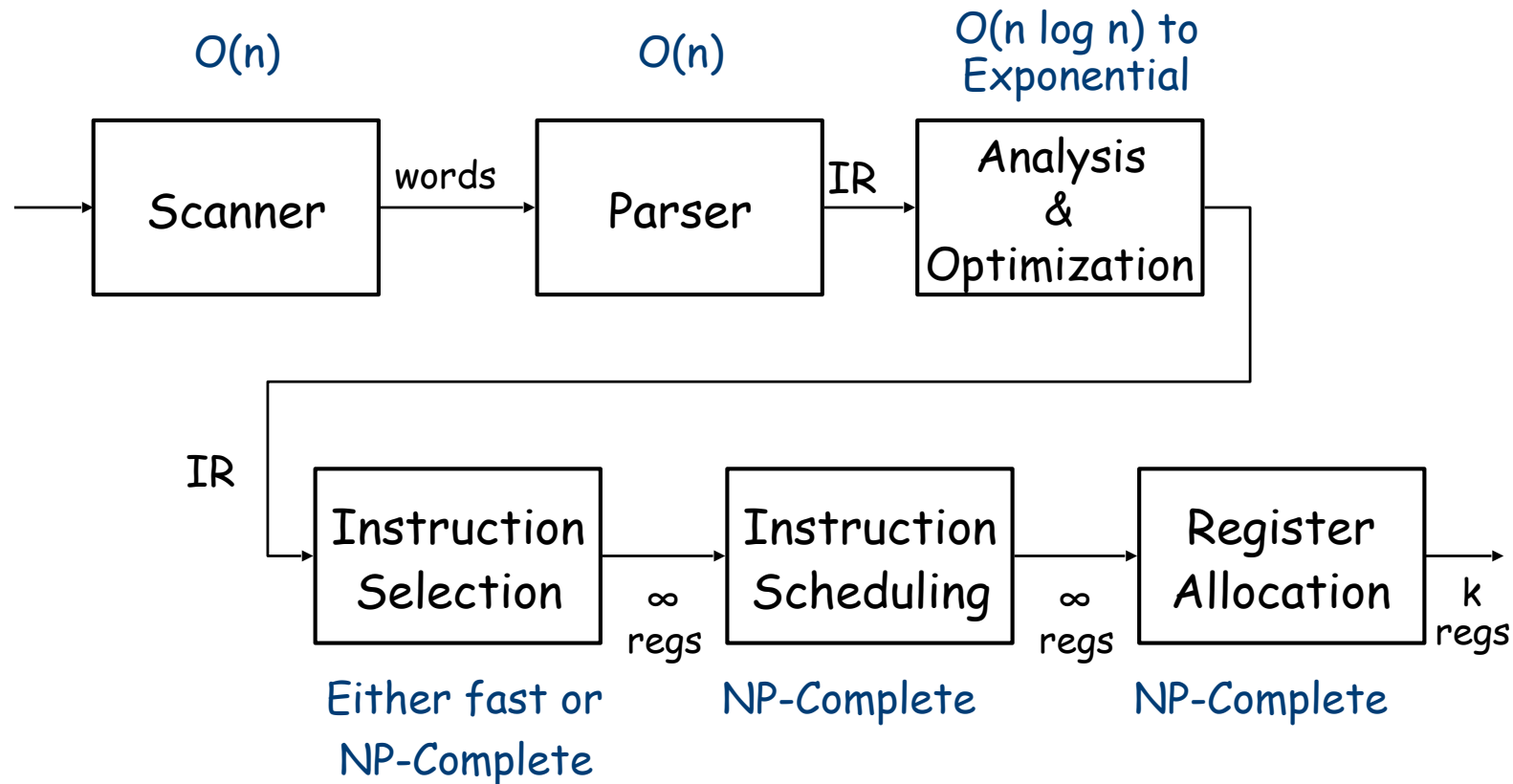


Introduction to Code Generation

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Structure of a Compiler

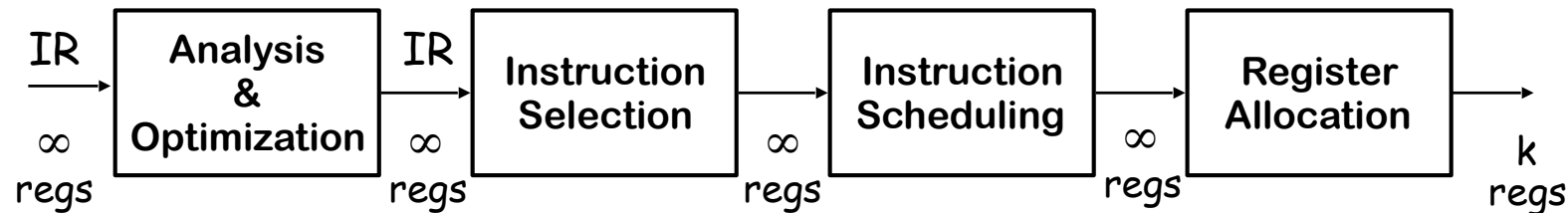


A compiler is a lot of fast stuff followed by some hard problems

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

Structure of a Compiler

We assume the following model



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

What about the IR ?

- Low-level, **RISC**-like IR such as **ILOC**
- Has "enough" registers
- **ILOC** was designed for this stuff with:
 - Branches, compares, & labels
 - Memory tags
 - Hierarchy of loads & stores
 - Provision for multiple ops/cycle

Analysis & Optimization

- The translation of the front end was obtained by considering the statements one of the time as they were encountered
- This initial IR contains general implementation strategies **that will work in any surrounding context**
- At run time the code will be executed in a more constrained and predictable context
- The optimizer analyses the IR form of the code to discover facts about the context and use them to rewrite (**transform**) the code so that it will compute the same answer in a **more efficient way**

The Back End

The compiler back end traverses the IR form and emits the code for the target machine

- It selects target-machine operations to implement each IR operation (**Instruction selection**)
- It chooses an order in which the operations will execute efficiently (**Instruction scheduling**)
- It will decide which values will reside in registers and which in memory (**Register allocation**)

Definitions

Instruction selection

- Mapping IR into assembly code
- Assumes a fixed **storage mapping** & **code shape**
- Combining operations, using address modes (instr. reg+offset or reg to reg mode)

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

These 3 problems
are tightly coupled
and need static
analysis

Definition

- The compiler must choose among many alternative ways to implement each construct on a given processor
- Those choices have a strong and direct impact on the quality of the final produced code
- Code shape is the end product of many decisions (big & small)


Impact

- Code shape has a strong impact on the behaviour of the compiled code and on the ability of the optimizer and back end to improve it
- Code shape can encode important facts, or hide them

Code Shape

Example -- the case statement on a character value

- Implement it as cascaded if-then-else statements
 - Cost depends on where your case actually occurs
 - $O(256)$
- Implement it as a binary search
 - Need a dense set of conditions to search
 - Uniform ($\log 256$) cost
- Implement it as a jump table
 - Lookup address in a table & jump to it
 - We trade data space for speed
 - Uniform (constant) cost



Performance depends on order of cases!

All these are legal (and reasonable) implementations of the switch statement

Which implementation for switch?

The one that is the best for a particular switch statement depends on many factors such as:

- The **number of cases** and their **relative executions frequencies**
- The knowledge of the cost structure for branching on the processor

Even when the compiler does not have enough information to choose it must choose an implementation strategy

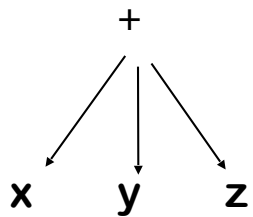
No amount of massaging or transforming will convert one into another

Code Shape: the ternary operation $x+y+z$

Several ways to implement $x+y+z$

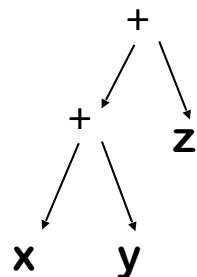
Addition is commutative & associative for integers

$x + y + z$



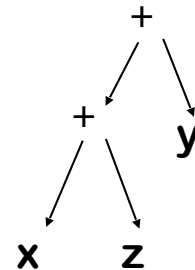
$x + y \rightarrow t1$

$t1 + z \rightarrow t2$



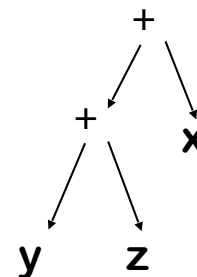
$x + z \rightarrow t1$

$t1 + y \rightarrow t2$



$y + z \rightarrow t1$

$t1 + x \rightarrow t2$



- What if the compiler knows that x is constant 2 and z is 3? The compiler should detect $2+3$ evaluates and fold it into the code
- 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

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

How to generate ILOC code

- The three-address form lets the compiler name the result of any operation and preserve it for later reuse
- It uses always new register and leave to the allocator the duty of reduce them
- To generate code for a trivial expression $a+b$ the compiler emits code to ensure that the values of a and b are in registers
- If a is stored in memory at offset $@a$ in the current Activation Record (AR), the code is

$$\begin{array}{lll} \text{loadI} & @a & \Rightarrow r_1 \\ \text{loadA0} & r_{arp}, r_1 & \Rightarrow r_a \end{array}$$

Generating Code for Expressions

the node of the AST

The idea

- Assume an **AST** as input and **ILOC** as output
- Use a **postorder treewalk** evaluator
 - 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()** and **val()**
- Works for simple expressions
- Easily extended to other operators

```
expr(node) {
  register 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 ← NextRegister();
      emit (loadl, offset(node), none, t2);
      result ← NextRegister();
      emit (loadAO, t1, t2, result);
      break;
    case NUMBER:
      result ← NextRegister();
      emit (loadl, val(node), none, result);
      break;
  }
  return result;
}
```

Generating Code for Expressions (a naive translation)

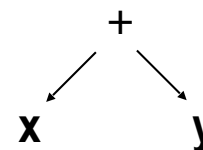
```

expr(node) {
  register 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 ← NextRegister();
      emit (loadI, offset(node), none, t2);
      result ← NextRegister();
      emit (loadAO, t1, t2, result);
      break;
    case NUMBER:
      result ← NextRegister();
      emit (loadI, val(node), none, result);
      break;
  }
  return result;
}

```

base(id) loads the right pointer to the AR where id is defined in register r_{arp}

Example:



Produces for register counter 0 :

espr("x"):

NextRegister(): r1 loadI @x -> r1

NextRegister(): r2 loadAO rarp, r1 -> r2

espr("y"):

NextRegister(): r3 loadI @y -> r3

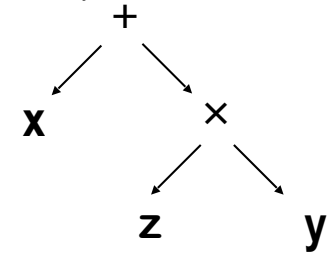
NextRegister(): r4 loadAO rarp, r3 -> r4

NextRegister() : r5

Emit(add, r2, r4, r5) :

add r2, r4 -> r5

Generating Code for Expressions (a naive translation)



```
expr(node) {
  register 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 ← NextRegister();
      emit (loadI, offset(node), none, t2);
      result ← NextRegister();
      emit (loadAO, t1, t2, result);
      break;
    case NUMBER:
      result ← NextRegister();
      emit (loadI, val(node), none, result);
      break;
  }
  return result;
}
```

Produces for register counter 0 :

espr("x"):

NextRegister():r1, loadI @x → r1

NextRegister():r2 loadAO rarp, r1 → r2

espr("z"):

NextRegister():r3 loadI @z → r3

NextRegister():r4 loadAO rarp, r3 → r4

espr("y"):

NextRegister():r5 loadI @y → r5

NextRegister():r6 loadAO rarp, r5 → r6

NextRegister():r7

Emit(mul, r4,r6,r7) :

mult r4, r6 → r7

NextRegister():r8

Emit(add, r2,r7,r8) :

add r2, r7 → r8

Effects of code shape on the demand of registers

- Code shape decisions encoded into the tree walk code generator have an effect on the demand of registers
- The previous naive code uses 8 registers + r_{arp}
- The register allocator (later in compilation) can reduce the demand for register to 3 + r_{arp}

```
loadI    @x    -> r1
loadA0   rarp, r1 -> r1
loadI    @z    -> r2
loadA0   rarp, r2 -> r2
loadI    @y    -> r3
loadA0   rarp, r3 -> r3
mult     r2,   r3 -> r2
add      r1,   r2 -> r2
```

The best solution: alternate right and left children

evaluating $z \times y$ first

<i>load</i>	@z	$\Rightarrow r_1$
<i>loadA0</i>	r_{arp}, r_1	$\Rightarrow r_2$
<i>load</i>	@y	$\Rightarrow r_3$
<i>loadA0</i>	r_{arp}, r_3	$\Rightarrow r_4$
<i>mult</i>	r_2, r_4	$\Rightarrow r_5$
<i>load</i>	@x	$\Rightarrow r_6$
<i>loadA0</i>	r_{arp}, r_6	$\Rightarrow r_7$
<i>add</i>	r_7, r_5	$\Rightarrow r_8$

General rule: evaluate first the child that has more demand for registers

Code shape!

after register allocation

<i>load</i>	@z	$\Rightarrow r_1$
<i>loadA0</i>	r_{arp}, r_1	$\Rightarrow r_1$
<i>load</i>	@y	$\Rightarrow r_2$
<i>loadA0</i>	r_{arp}, r_2	$\Rightarrow r_2$
<i>mult</i>	r_1, r_2	$\Rightarrow r_1$
<i>load</i>	@x	$\Rightarrow r_2$
<i>loadA0</i>	r_{arp}, r_2	$\Rightarrow r_2$
<i>add</i>	r_2, r_1	$\Rightarrow r_1$

Some observations

What if our IDENTIFIER is

- already in a register?
- in a global data area?
- a parameter value?
 - * call by value
 - * call by reference

Extending the Simple Treewalk Algorithm

It assumes a single case for id, more 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 ?
 - Call-by-value \Rightarrow it can be handled as it was a local variable as before
 - Call-by-reference \Rightarrow extra indirection 3 instructions. The value may not be kept in a register across an assignment (see next slide)
- What about function calls in expressions?
 - Generate the calling sequence & load the return value
 - Severely limits compiler's ability to reorder operations

Keeping values in registers

- In a register-to register memory model, the compiler tries to assign many values as possible to virtual registers
- Then the register allocator will map the set of virtual to physical registers inserting the spills
- However, the compiler can keep values in a register only for **unambiguous value**:
a value that can be accessed with just one name is unambiguous

The problem with ambiguous values

- Consider a and b ambiguous and the following code

a := m+n;

b := 13;

c:= a+b;

If a and b refers to the same location c gets value 26,
otherwise c gets value m+n+13;

The compiler cannot keep a in a register during the assignment of b unless it proves that the set of location that the two name refer to are disjoint. This analysis can be expensive!

sharing analysis !

Where do ambiguous values arise?

Ambiguous values may arise in several ways :

- values stored in a pointer based variable
- call by reference formal parameter
- many compilers treat array element values as ambiguous because they can not tell if two references $A[i,j]$ e $A[n,m]$ refer to the same location

for safety the compiler has to consider that values as **ambiguous**

Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls (exp. and trig fun.)

Mixed-type expressions

- Insert conversion code as needed from conversion table
- Most languages have symmetric & rational conversion tables

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

If the type cannot be inferred at compile time, the compiler must insert code for run-time checks that test for illegal cases!

Extending the Simple Treewalk Algorithm

What about evaluation order?

Can use commutativity & associativity to improve code for integers

- For recognising that already computed that value

$$a+b = b+a$$

- For recognising that it can compute subexpressions

$$a+b+d \text{ and } c+a+b$$

(it does not if it evaluates the expressions in strict left right order!)

It should not reorder floating point expressions!

- The subset of reals represented on a computer does not preserve associativity

$a-b-c$ the results may depend on the evaluation order!

Handling Assignment

(just another operator)

lhs ← *rhs*

Strategy

- Evaluate rhs to a **value** (an rvalue)
- Evaluate lhs to a **location** (an lvalue)
 - lvalue is a register ⇒ move rhs
 - lvalue is an address ⇒ 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

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 when Reference (pointer) Counts is used

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

- 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

- Useful in real-time applications, user interfaces
- Counts will be in cache

Disadvantages

- Freeing root node of a graph implies a lot of work & disruption
- Cyclic structures pose a problem

Handling Assignment when Reference Counts is used

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 (lhs→count = 0)
    free lhs
```

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

Summary

Code Generation for Expressions

- Simple treewalk produces reasonable code
 - Execute most demanding subtree first
 - Can implement treewalk explicitly, with an Attributed grammar or ad hoc Syntax directed translation ...
- Handle assignment as an operator
 - Insert conversions according to language-specific rules
 - If compile-time checking is impossible, check tags at runtime
 - Talked about how to handle reference counting (an alternative to Garbage Collector)

Next computing Array access!