The structure of a compiler

Reality

![Diagram of compiler stages: scan, parse, optimize, select, schedule, allocate, generating machine code.]

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

Code generation

Chapters 8 and 9

- really deal with the same set of problems
- lectures will skip back and forth between them

Compilation model

- low-level, RISC-like intermediate language
- separate selection, scheduling, and allocation
- assume a sufficient number of registers in early phases

\[
\begin{align*}
& I_R^1 \xrightarrow{\text{optimize}} I_R^2 \xrightarrow{\text{select}} \xrightarrow{\text{schedule}} \xrightarrow{\text{allocate}} \text{asm} \\
& \text{regs} \to \text{regs} \to \text{regs} \to \text{regs} \\
& k \to k \to k \to k
\end{align*}
\]

⇒ select is fairly simple
⇒ allocate and schedule are complex

Code generation

AS&U think of intermediate code generation and machine code generation as two problems

- in reality, the differences can be large or small
  - AST→RISC instructions (very different)
  - quads→VAX instructions (quite similar)
- changes in machine and compiler architecture make the distinction unimportant

Superscalar instruction issue \(\Rightarrow\) dominant issues

- sequencing instructions (scheduling)
- combining instructions (multiple issue)
- keeping values in registers (register allocation)

Definitions

Instruction selection

- the process of mapping \(I_R^1\) into \(I_R^2\) (3 addr. code)
- assumes a fixed storage mapping for all variables
- combining instructions, using address modes

Register allocation

- the process of deciding which values reside in registers
- may change storage mapping (insert spill code)

Instruction scheduling

- the process of reordering instructions to hide latencies
- assumes a fixed program
- may change demand for registers

Of course, the problems are very inter-related.
The big picture

How hard are these problems?

Instruction selection

- can make locally optimal choices
- can automate construction

Register allocation

- one basic block: (no spilling)
  - one register size ⇒ linear time
  - two register sizes ⇒ NP-complete
- whole procedure: NP-complete (no spilling)

Instruction scheduling

- basic block ⇒ polynomial time
- across blocks ⇒ extremely hard

Before looking at code-generator generators (CGG), look at code generators (CG)

Code generation

1. Source code → intermediate representation

   (a) storage layout
   (b) code for simple expressions
   (c) code for control structures
   (d) code for procedure calls
   (e) code for complex expressions
   (f) better code for expressions

2. Intermediate representation → target code

   (a) instruction selection
   (b) instruction scheduling
   (c) register allocation

We will be covering each issue in order

Intermediate representation

We’ll be targeting RISC-like processors

- load-store architecture
- register-transfer language
- three-address code
- explicit loads and stores

Examples

- load r1, <addr> $ r1 ← value at <addr>
- loadi r1, <const> $ r1 ← value of <const>
- store r1, <const> $ <addr> ← r1
- move r1, r2 $ r1 ← r2
- add r1, r2, r3 $ r1 ← r2 + r3
- sub r1, r2, r3 $ r1 ← r2 - r3
- mult r1, r2, r3 $ r1 ← r2 * r3
- jmp <addr> $ jump to <addr>

Conventional wisdom says that we can (and should) attack each of these problems independently

Instruction selection

- use either tree-matching, or instruction matching
- either:
  1. assume “enough” registers, or
  2. target “important” values into registers
- determines shape of the code

Register allocation/assignment

- virtual registers ⇒ map into real (physical) registers; typically two steps:
  1. What virtual register ⇒ allocation
  2. Which real register ⇒ assignment
- targeting ⇒ prioritize by “benefit” metric

Instruction scheduling

- within a block, use list-scheduling
- across blocks (small loops), use software pipelining

Note the large number of “fuzzy” terms!
Storage layout

Local, non-static storage
- stash them in the frame
- keep frames on the stack
- assign offsets from the frame pointer
- pad for word alignment

Global or static storage
- offset from procedure’s static data area (static)
- offset from known global label (global)

Varying sized storage
- local, non-static ⇒ top of stack frame
- other cases ⇒ allocate on the heap

Simple expressions

Expression trees:
- adopt a simple treewalk scheme
- assign a virtual register to each operator
- emit code in postorder walk

Support routines:
- base( str ) — returns the name of a new virtual register that contains the base address for str and emits the corresponding code
- offset( str ) — returns the name of a new virtual register that contains the offset of str and emits the corresponding code
- newtemp() — returns a new virtual register name

Assume:
- assume tree reflects precedence, associativity
- assume all operands are integers

Key Issue
- what variables can be safely allocated to registers?
- what variables should be allocated to registers?

Encoding the decision
- assign some variables to virtual registers
- treat those that cannot be in a register carefully

Depends on the philosophy of the register allocator

Simple expressions

expr( node ) : int
int result, t1, t2, t3;
switch( type of node )
case PLUS:
t1 = expr( left child of node );
t2 = expr( right child of node );
result = newtemp();
emit( add, result, t1, t2 );
break;
case ID:
t1 = base( node.val );
t2 = offset( node.val );
t3 = newtemp();
emit( add, t3, t1, t2 );
result = newtemp();
emit( load, result, t3 );
break;
case NUM:
result = newtemp();
emit( loadi, result, node.val );
break;
return result
Control structures

Basic blocks
- a basic block is a sequence of straight line code
- if one instruction executes, they all execute
- a maximal sequence of instructions without branches
- a label starts a new basic block

Early work in code optimization focused on basic blocks.
- common subexpression elimination
- constant folding
- “optimal” code generation
- list scheduling

Assignment statement
\[ \text{lhs} \leftarrow \text{rhs} \]

Strategy
- evaluate \( \text{rhs} \) to a value \((an\ \text{rvalue})\)
- evaluate \( \text{lhs} \) to an address \((an\ \text{lvalue})\)
  \[ \begin{align*}
  & \text{i) lvalue is register } \Rightarrow \text{move it} \\
  & \text{ii) lvalue is address } \Rightarrow \text{store it}
  \end{align*} \]

Registers versus memory
- non-aliased scalars \( \Rightarrow \) can go in a register
- aggregate or potentially aliased \( \Rightarrow \) in memory

Overview
- control flow links up the basic blocks
- ideas are simple
- implementation requires some bookkeeping
- some care is required for good code
- \text{design-time} vs. \text{compile-time} vs. \text{run-time}

Early optimizing compilers generated good code for basic blocks and linked them together carefully.
Control structures

if-then-else
1. evaluate the expression to true or false
2. if true, fall through to then part
   branch around else part
3. if false, branch to else part
   fall through to next statement

Example
r1 ← expr if not(r1) br L1... br L2
L1: ...
L2: ...
evaluate the expression
compare and branch
stms for then part
branch to exit
stms for else part
following stmt

Control structures

while loop or do loop
1. evaluate the control expression
2. if false, branch beyond end of loop
   if true, fall through into loop body
3. at end, re-evaluate the control expression
4. if true, branch to top of loop body
   if false, fall through

Example
r1 ← expr if not(r1) br L2
L1: ...
r1 ← expr if r1, br L1
L2: ...
evaluate the expression
compare and branch
loop body
stms for then part
branch to exit
stms for else part
following stmt

Test at end ⇒ simple loop is one block

Control structures

case statement
1. evaluate the controlling expression
2. branch to the selected case
3. execute its code
4. branch to the following statement

Recall — Procedure abstraction

The essentials:
• on entry, establish p’s environment
• at a call, preserve p’s environment
• on exit, tear down p’s environment
• in between, addressability and proper lifetimes

Key issue:
⇒ finding the right case

<table>
<thead>
<tr>
<th>Method</th>
<th>When</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear search</td>
<td>few cases</td>
<td>O(</td>
</tr>
<tr>
<td>binary search</td>
<td>sparse</td>
<td>O(log(</td>
</tr>
<tr>
<td>jump table</td>
<td>dense</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

procedure p

procedure q
Procedure calls

Code for procedure calls

- save registers
- extend basic frame
- find static data area
- initialize locals
- evaluate & store params.
- allocate child’s frame
- store FP and RA
- set FP for child
- jump to child

RA:
- copy return value
- restore params.
- free child’s frame
- store return value
- unextend basic frame
- restore registers
- restore parent’s FP
- jump to RA

prolog code
- for local data
- if needed

start of a call
in child’s frame
may handle RA
post-call code
if needed
epilog code
hardware ret

† FP is frame pointer, RA is return address

Issues

- caller saves or callee saves
- no need for a static data area
- parameters on stack or in registers
- return address in stack or register
- where’s the frame pointer
- static links, frame display, or global display
- return value on stack or in register

The first test of a linkage convention is that it works

Next lecture

Code generation

- array references, function calls, mixed type
  expressions, boolean expressions
- DAG construction

Please read ASU Chapters 8.4-8.7 & 5.2/9.8