

Optimizations...

- intermediate code:
 - pros: machine independent, optimization opportunities which will not need to be retargeted for another platform
 - cons: yet another language
 - possible representations:
 - trees, three-address code ($y := x \text{ op } z$) (or quadruples), triple, polish notation...

- Optimization seeks to improve a program's utilization of some resource
 - Execution time (most often)
 - Code size
 - Network messages sent
 - Battery power used, etc.
- Optimization should not alter what the program computes
 - The answer must still be the same
- What to optimize:
 - cost/performance argument (opt-s hard to implement, increase compilation time...)
 - code to consider based on number of: occurrences in code, repetitions of execution, uses of compiled code

Optimization options

- Apply *local optimizations* in basic blocks
 - single entry/exit point
 - preserve outcome of computation
 - select different representation for sequence in block
- *Global optimization* – across CFG (trace) of BBs
 - harder to prove
 - use flow analysis
- *Intra-procedural optimizations*

Local optimizations

- Algebraic simplification
 - Some statements can be deleted:
 $x := x + 0$
 $x := x * 1$
 - Some statements can be simplified:
 $x := x * 0 \quad \Rightarrow \quad x := 0$
 $y := y ** 2 \quad \Rightarrow \quad y := y * y$
 $x := x * 8 \quad \Rightarrow \quad x := x \ll 3$
 $x := x * 15 \quad \Rightarrow \quad t := x \ll 4; x := t - x$

Constant Folding

- Operations on constants can be computed at compile time
 - (algebraic simplifications)
- In general, if there is a statement
 $x := y \text{ op } z$
 - And y and z are constants
 - Then $y \text{ op } z$ can be computed at compile time
- Example: $x := 2 + 2 \Rightarrow x := 4$
- Example: $\text{if } 2 < 0 \text{ jump } L$ can be deleted

- In general –
 - values of expressions involving compiler time constants – replace unary/binary tree with a constant node with corr. value
 - operations involving constants as operand – replace binary operation subtree with its non-constant subtree

Constants within blocks

- traverse values from constant assignments within a basic block
- when an assignment involves a constant rhs put the target in a “known vars” list (symbol table)
- whenever a var is accessed, check to see if its in “known vars” list, and if constant, then replace node in AST
- remove variables from the list when a non-constant assignment is encountered, or at the end of a basic block
- can use “known list” to track accesses and delete useless variables

Constants across blocks

- why does a block end:
 - entrance to selection/looping statement
 - exit from same
 - calls to procs/funcs
- when can we retain value of variable?

Flow of Control Optimizations

- Eliminating unreachable code:
 - Code that is unreachable in the control-flow graph
 - Basic blocks that are not the target of any jump or “fall through” from a conditional
 - Such basic blocks can be eliminated
- Removing unreachable code makes the program smaller
 - And sometimes also faster
 - Due to memory cache effects (increased spatial locality)

Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment
 - Intermediate code can be rewritten to be in single assignment form
- ```
x := z + y b := z + y
a := x => a := b
x := 2 * x x := 2 * b
 (b is a fresh register)
```
- More complicated in general, due to loops

## Common Subexpression Elimination

- Assume
  - Basic block is in single assignment form
  - A definition  $x :=$  is the first use of  $x$  in a block
- All assignments with same rhs compute the same value
- Example:

```
x := y + z x := y + z
... ...
w := y + z w := x
(the values of x, y, and z do not change in the ... code)
```

## Copy Propagation

- If  $w := x$  appears in a block, all subsequent uses of  $w$  can be replaced with uses of  $x$
- Example:

```
b := z + y b := z + y
a := b => a := b
x := 2 * a x := 2 * b
```
- This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination

## Copy Propagation and Constant Folding

- Example:

```
a := 5 a := 5
x := 2 * a ⇒ x := 10
y := x + 6 y := 16
t := x * y t := x << 4
```

## Copy Propagation and Dead Code Elimination

If

$w := rhs$  appears in a basic block  
 $w$  does not appear anywhere else in the program

Then

the statement  $w := rhs$  is dead and can be eliminated  
– Dead = does not contribute to the program's result

Example: (a is not used anywhere else)

```
x := z + y b := z + y b := z + y
a := x ⇒ a := b ⇒ x := 2 * b
x := 2 * a x := 2 * b
```

## Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimizations enables other opt.
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - The optimizer can also be stopped at any time to limit the compilation time

## An Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

## An Example

- Algebraic optimization:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

## An Example

- Algebraic optimization:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

## An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

## An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

## An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

## An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

## An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

## An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

## An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

## An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

## An Example

- Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

## An Example

- Dead code elimination:

```
a := x * x

f := a + a
g := 6 * f
```

- This is the final form

## If statements (CJUMP)

- if cond then e1 else e1 => e1
- if 0 then e1 else e2 => e2
- reorder e1/e2 to fall through on branch (follow by false label, do !cond if necessary)
- while 2<3 – dead code elimination
- (last class slide)

## Arrays and loops

- subscript expressions
  - occur frequently – good code important
  - precompute whatever possible
  - occur often within loops =>...
- ‘unroll loops’ (modulo unrolling)
  - amortize overhead of loop instructions over several loop executions
- branches to branches
- use of registers – e.g., loop index in reg.

## Local Optimizations. Notes.

- Intermediate code is helpful for many optimizations
- Many simple optimizations can still be applied on assembly language
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term

## Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code
  - They are target independent
  - But they can be applied on assembly language also
- Peephole optimization is an effective technique for improving assembly code
  - The “peephole” is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster) (looks for patterns)

## Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules
$$i_1, \dots, i_n \rightarrow j_1, \dots, j_m$$
where the rhs is the improved version of the lhs
- Example:  
`move $a $b, move $b $a → move $a $b`
  - Works if `move $b $a` is not the target of a jump
- Another example  
`addiu $a $a i, addiu $a $a j → addiu $a $a i+j`
- explicit addr computation instead of using addressing modes, shifts in stead of mul...

## Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast as peephole optimizations
  - Example: `addiu $a $b 0 → move $a $b`
  - Example: `move $a $a →`
  - These two together eliminate `addiu $a $a 0`
- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect

## SPIM and MIPS

- recommended to target Tiger compiler for SPIM ‘platform’ – MIPS32 [simulator](#)
- MIPS assembly – extended version of machine assembly – implements a virtual machine
  - hides complexities – delayed branches, loads, restricted address mode...
  - pseudoinstructions used by programmer/compiler, assembler translates these into appropriate sequence of machine instructions
  - we saw registers & calling convention already
  - runtime for Tiger ‘system calls’ – print, initArray... (unresolved symbols)
- documentation, download, and compilation inst. on webpage (SPIM and Tiger runtime)