CS536

Basic Blocks Optimization & Register Allocation

A Sahu

CSE, IIT Guwahati

Outline

- Transformation on Basic Block
- Peep Hole Optimization or Window optimization
- Register Allocation

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

```
a  := c * a
b  := 0
```

```
a := c*a
b := 0
```

Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed **across basic blocks**
- *Local transformations* are only performed on **single basic blocks**
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Common-Subexpression Elimination

- Remove redundant computations

```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

```
a1  := b0 + c0
b1  := a1 - d0
c  := b1 + c0
d  := a1 - d0
```

```
a1  := b0 + c0
b1  := a1 - d0
c  := b1 + c0
d  := b1
```

```
t1  := b * c
t2  := a - t1
t3  := b * c
t4  := t2 + t3
```

```
t1  := b * c
t2  := a - t1
t4  := t2 + t1
```

Dead Code Elimination

- Remove unused statements

```
b := a + 1
a := b + c
…
```

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)

```
if true goto L2
```

```
b := x + y
…
```

Remove unreachable code

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

→

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

Interchange of Statements

- Independent statements can be reordered

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1 := a - a
t2 := b + t1
t3 := 2 * t2
```

```
t1 := 0
t2 := b
t3 := t2 << 1
```

Peephole Optimization

- Examines a short sequence of target **instructions in a window** (**peephole**) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Opt: Eliminating Redundant Loads and Stores

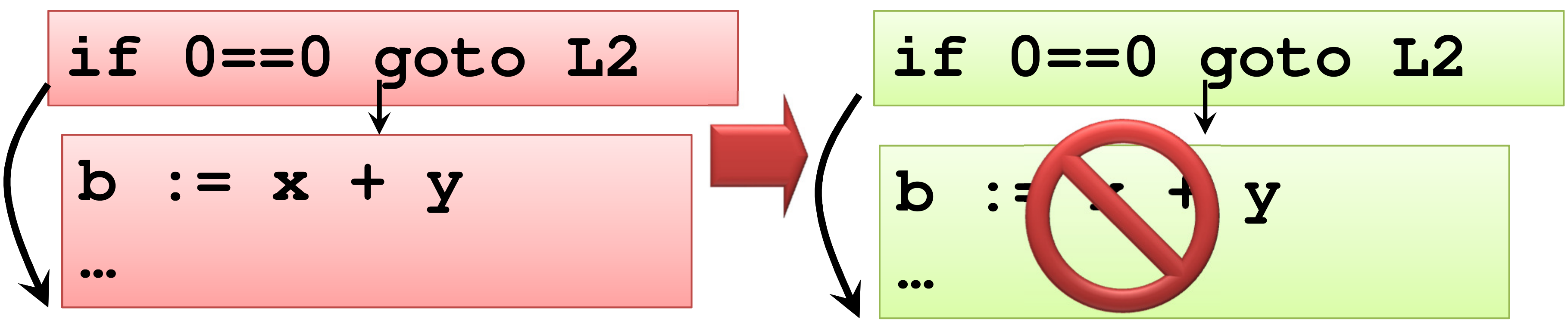
- Consider

```
MOV R0,a
MOV a,R0
```

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**)=false

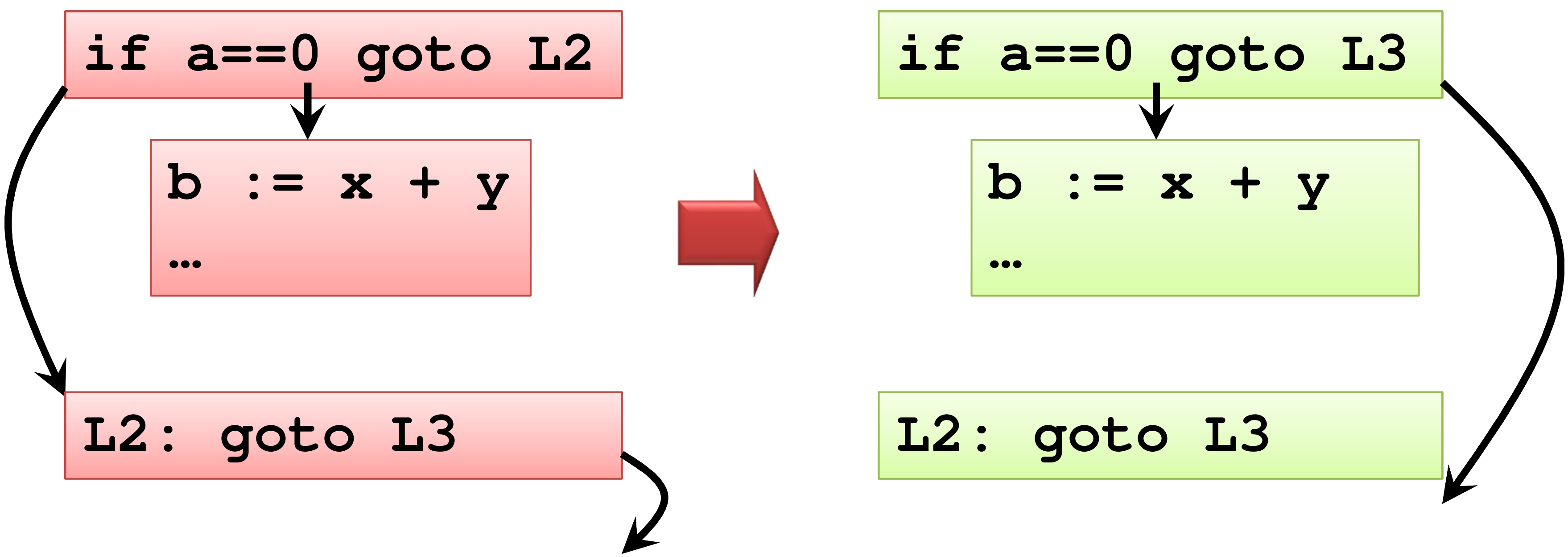
Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed



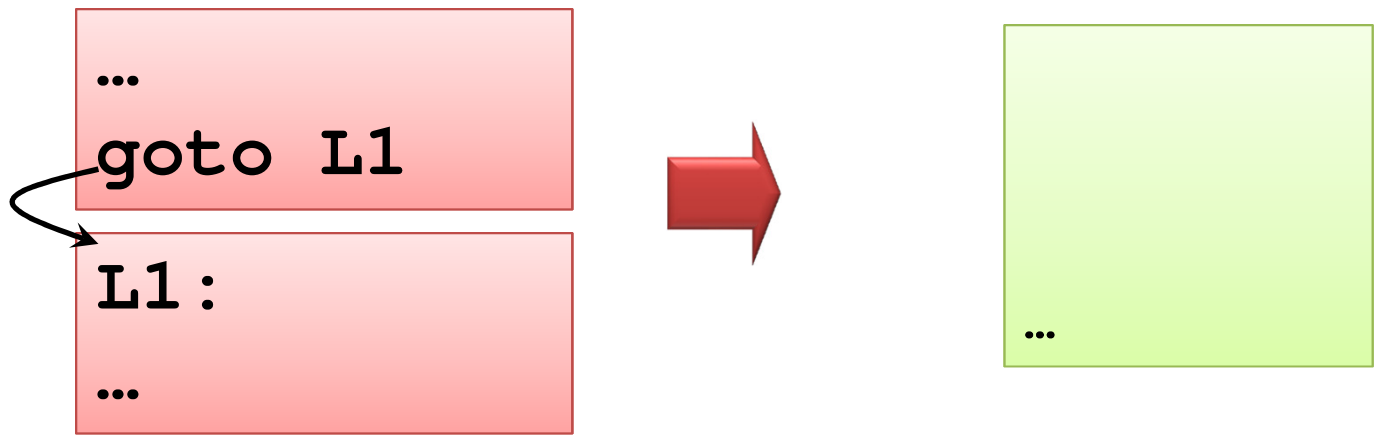
Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels



Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps



Other Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

```
…
a := x ^ 2
b := y / 8
```

```
…
a := x * x
b := y >> 3
```

Other Peephole Optimizations

- Utilize machine idioms

```
…
a := a + 1
```

→

```
…
inc a
```

- Algebraic simplifications

```
…
a := a + 0
b := b * 1
```

→

```
…
```

Register Allocation and Dead code Elimination

Register Allocation

How to best use the bounded number of registers.

- Reducing load/store operations
- What are best values to keep in registers?
- When can we ‘free’ registers?

Complications:

- special purpose registers
- operators requiring multiple registers.

Register Allocation Algorithms

- Local (basic block level):
  - **Basic - using liveness information**
  - **Register Allocation using graph coloring**
- Global (CFG)
  - Need to use global liveness information

Basic Code Generation

- Deal with each basic block individually.
- Compute liveness information for the block.
- Using liveness information, generate code that uses registers as well as possible.
- At end, generate code that saves any live values left in registers.

Concept: Variable Liveness

- For some statement *s*, variable *x* is **live** if
  - there is a statement *t* that uses *x*
  - there is a path in the CFG from *s* to *t*
  - there is no assignment to *x* on some path from *s* to *t*
- **A variable is *live* at a given point in the source code if it could be used before it is defined.**
- Liveness tells us whether we care about the value held by a variable.

Life and Next Use
Example I

Next-Use

- **Next-use** information is needed for **dead-code elimination and register assignment**
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  $$i\text{: } x := y \text{ op } z$$

  - Add liveness/next-use info on *x*, *y*, and *z* to statement *i*
  - Set *x* to "not live" and "no next use"
  - Set *y* and *z* to "live" and the next uses of *y* and *z* to *i*

Next-Use (Step 1)

```
i: a := b + c

j: t := a + b
```

[*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live

Next-Use (Step 2)

```
i: a := b + c

j: t := a + b
```

live(**a**) = true	*nextuse*(**a**) = *j*
live(**b**) = true	*nextuse*(**b**) = *j*
live(**t**) = false	*nextuse*(**t**) = none

[*live*(**a**) = true, *live*(**b**) = true,
live(**t**) = true, *nextuse*(**a**) = none,
nextuse(**b**) = none,
nextuse(**t**) = none]

Compute live/next-use information at *j*

Next-Use (Step 3)

`i: a := b + c`

[*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false, *nextuse*(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

`j: t := a + b`

[*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information to *i*

Next-Use (Step 4)

live(**a**) = false *nextuse*(**a**) = none
live(**b**) = true *nextuse*(**b**) = *i*
live(**c**) = true *nextuse*(**c**) = *i*
live(**t**) = false *nextuse*(**t**) = none

i: `a:= b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false, *nextuse*(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: `t:= a + b` [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *i*

Life and Next Use
Example II

Example: When is a live?

```
a := b + c
t1 := a * a
b := t1 + a
c := t1 * b
t2 := c + b
a := t2 + t2
```

a is live

Assume a,b and c are used after this basic block

Example: When is b live?

```
a := b + c
t1 := a * a
b := t1 + a
c := t1 * b
t2 := c + b
a := t2 + t2
```

b is live

b is live

Assume a,b and c are used after this basic block

Computing live status in basic blocks

Input: A basic block.

Output: For each statement, set of live variables

1. Initially all non-temporary variables go into live set (L).
2. for i = *last* statement to *first* statement:

   for statement i: x := y op z

   1. Attach L to statement i.
   2. Remove x from set L.
   3. Add y and z to set L.

Example

live = {

a := b + c

live = {

t1 := a * a

live = {

b := t1 + a

live = {

c := t1 * b

live = {

t2 := c + b

live = {

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {b,c} ← what does this mean???

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Graph Coloring

- The **vertex/default** coloring of a graph G = (V,E) is a mapping C: V→ S, where S is a finite set of colors, such that if edge vw is in E, **C(v) != C(w).**
- Problem is NP (for more than 2 colors) ➔ no polynomial time solution.
- Fortunately there are approximation algorithms.
- Greedy Algo:
  - Order vertices in any order,
  - choose color for v_i with smallest available color not used by v_i neighbors among v_1 to v_{i-1}, otherwise choose a new color for v_i