Code Optimization

Introduction

- Criterion of code optimization
  - Must preserve the semantic equivalence of the programs
  - The algorithm should not be modified
  - Transformation, on average should speed up the execution of the program
  - Worth the effort: Intellectual and compilation effort spend on insignificant improvement.

    Transformations are simple enough to have a good effect

Themes behind Optimization Techniques

- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
- Extract more information about code: More info – better code generation

Redundancy elimination

- Redundancy elimination = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Value numbering
    - Associates symbolic values to computations and identifies expressions that have the same value
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.
  - Partial redundancy elimination
    - Inserts computations in paths to convert partial redundancy to full redundancy.

Optimizing Transformations

- Compile time evaluation
- Common sub-expression elimination
- Code motion
- Strength Reduction
- Dead code elimination

Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

Compile-Time Evaluation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
```

↓

```
area := 3.14286 * r ** 2
```

Compile-Time Evaluation

- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.
- Example:

```
pi := 3.14286
area = pi * r ** 2
```

```
                area = 3.14286 * r ** 2
```

Common Sub-expression Evaluation

- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

Example:

```
a := b * c                a := b * c
…
…                         …
x := b * c + 5            x := a + 5
```

Code Motion

- Moving code from one part of the program to other without modifying the algorithm
  - Reduce size of the program
  - Reduce execution frequency of the code subjected to movement

Code Motion

1. *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size.

Example: Code hoisting

```
if (a< b) then
      z := x ** 2
else
      y := x ** 2 + 10
```

```
      temp : = x ** 2
if (a< b) then
            z := temp
else
            y := temp + 10
```

“x ** 2“ is computed once in both cases, but the code size in the second case reduces.

Code Motion

- Move expression out of a loop if the evaluation does not change inside the loop.

Example:

```
while ( i < (max-2) ) ...
```

Equivalent to:

```
t :=  max - 2
while ( i < t ) ...
```

Strength Reduction

- Replacement of an operator with a less costly one.

Example:

```
for i=1 to 10 do
    …
    x = i * 5
    …

    end
```

→

```
    temp = 5;
for i=1 to 10 do
…
x = temp
…
    temp = temp + 5
end
```

Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point -> its value is not used anywhere in the program
  - An assignment is dead -> assignment assigns a value to a dead variable

Dead Code Elimination

```r
foo <- function() {
  a <- 24
  if (a > 25) {
    return(25)
    a <- 25 # dead code
  }
  return(a)
  b <- 24 # dead code
  return(b) # dead code
}
```

```r
foo <- function() {
  a <- 24
  if (a > 25) {
    return(25)
  }
  return(a)
}
```

Copy Propagation

- What does it mean?
  - Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y.
- When is it performed?
  - At any level, but usually early in the optimization process.
- What is the result?
  - Smaller code

Copy Propagation

- `f := g` are called copy statements or copies
- Use of `g` for `f`, whenever possible after copy statement

Example:

x[i] = a;	x[i] = a;
sum = x[i] + a;	sum = a + a;

- May not appear to be code improvement, but opens up scope for other optimizations.

Loop Optimization

- Decrease the number if instruction in the inner loop
- Even if we increase no of instructions in the outer loop
- Techniques:
  - Code motion
  - Induction variable elimination
  - Strength reduction

Induction variable elimination

```
int j = 0;
for (int i = 0; i < 100; i++) {
    j = 2*i;
}
return j;
```

```
int j = 0;
int s = 0; //2*i when i == 0
for (int i = 0; i < 100; i++) {
  j = s;
  s = s + 2; //+2 since i gets incremented by 1 each iteration
}
```

Peephole Optimization

- Pass over generated code to examine a few instructions, typically 2 to 4
  - Redundant instruction Elimination: Use algebraic identities
  - Flow of control optimization: removal of redundant jumps
  - Use of machine idioms

Peephole Optimization

- Constant Folding

  `x := 32` becomes `x := 64`

  `x := x + 32`

- Unreachable Code

  `goto L2`

  `x := x + 1` □ unneeded

- Flow of control optimizations

  `goto L1` becomes `goto L2`

  …

  `L1: goto L2`

Peephole Optimization

- Algebraic Simplification

  `x := x + 0` □ unneeded

- Dead code

  `x := 32` □ where x not used after statement

  `y := x + y` □ `y := y + 32`

- Reduction in strength

  `x := x * 2` □ `x := x + x`

Optimization of Basic Blocks

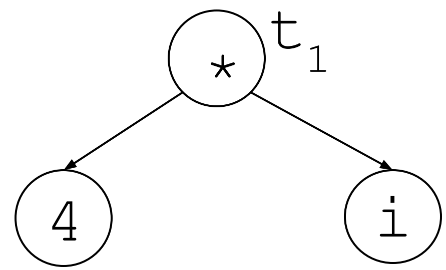
- Many structure preserving transformations can be implemented by construction of DAGs of basic blocks

DAG representation of Basic Block (BB)

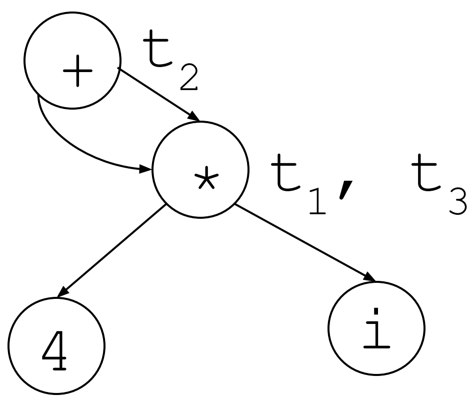
- Leaves are labeled with unique identifier (var name or const)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (identifiers)
- Edges relates operands to the operator (interior nodes are operator)
- Interior node represents computed value
  - Identifier in the label are deemed to hold the value

Example: DAG for BB

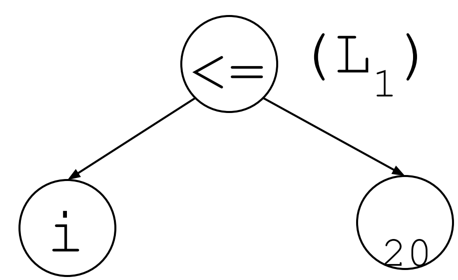
$t_1 := 4 * i$



$t_1 := 4 * i$
$t_3 := 4 * i$
$t_2 := t_1 + t_3$



if (i <= 20)goto L_1

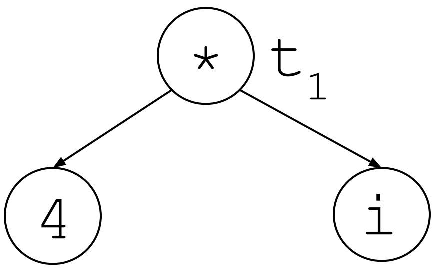


Construction of DAGs for BB

- I/p: Basic block, *B*
- O/p: A DAG for *B* containing the following information:
  1) A label for each node
  2) For leaves the labels are ids or consts
  3) For interior nodes the labels are operators
  4) For each node a list of attached ids (possible empty list, no consts)

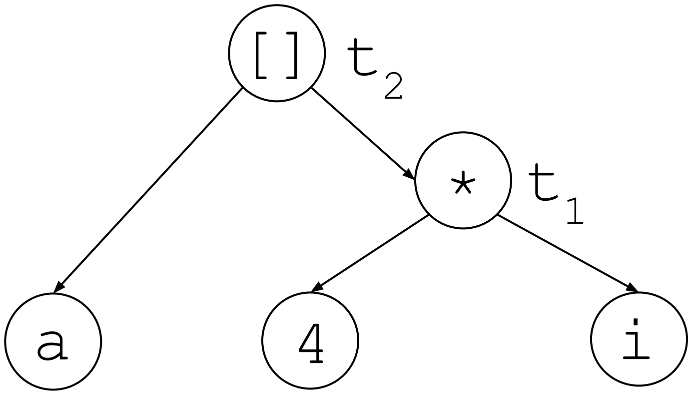
Example: DAG construction from BB

t_1 := 4 * i



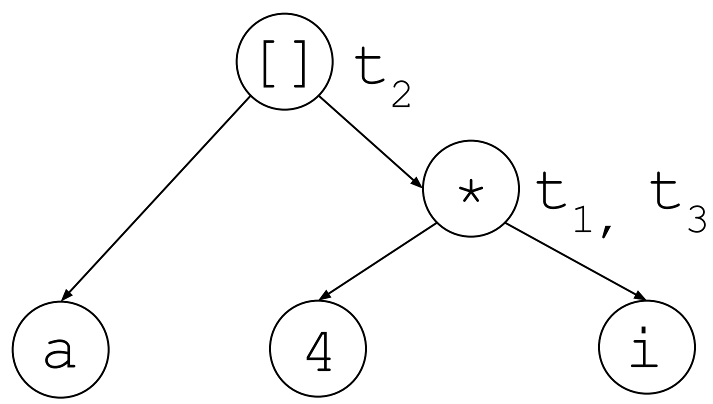
Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
```



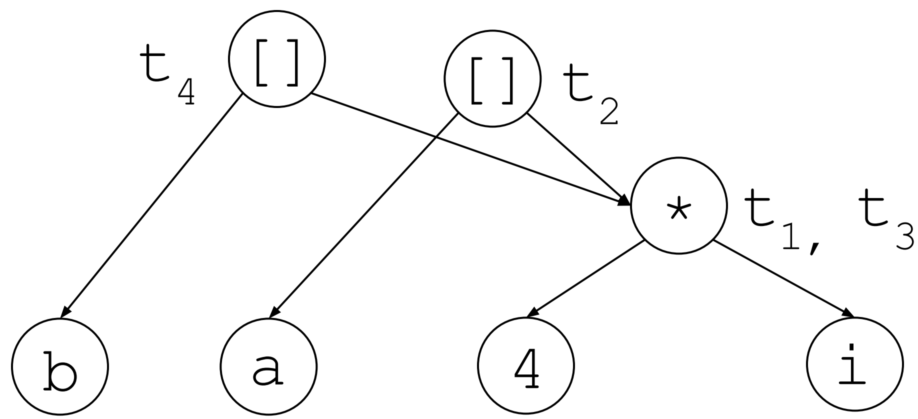
Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
```



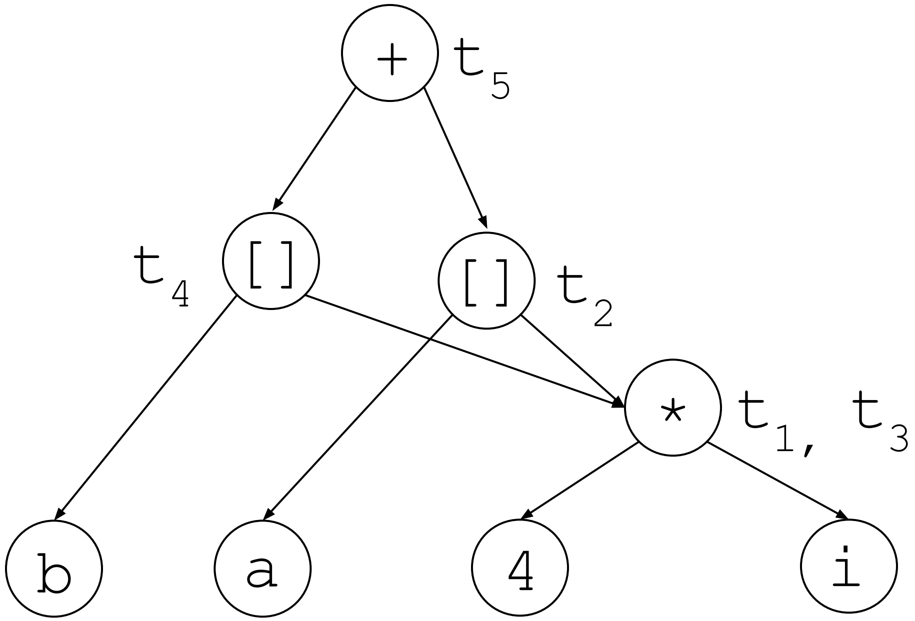
Example: DAG construction from BB

```
t1  := 4 * i
t2  := a [ t1 ]
t3  := 4 * i
t4  := b [ t3 ]
```



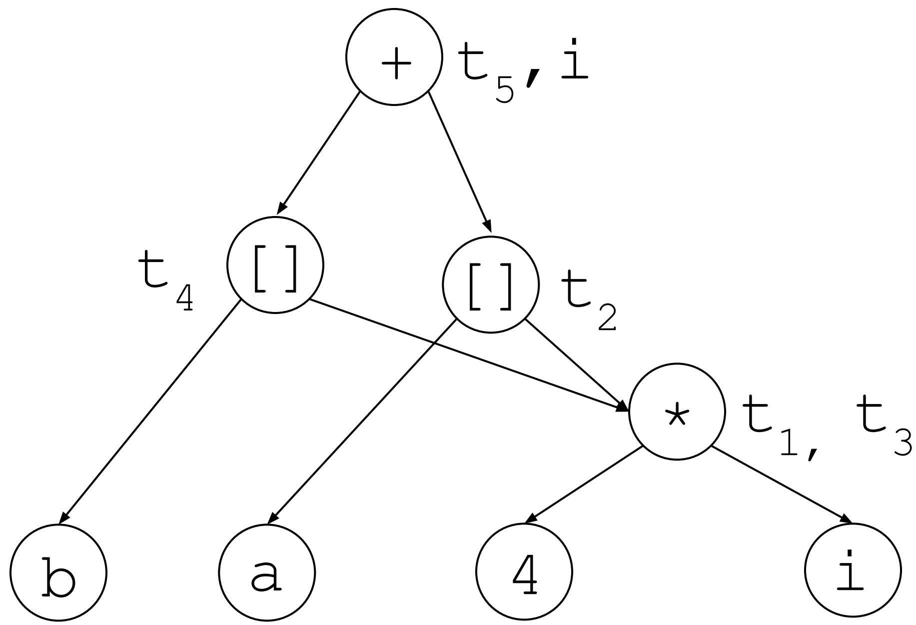
Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 + t4
```



Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 + t4
i := t5
```



Optimization of Basic Blocks

- Common sub-expression elimination: by construction of DAG
  - Note: for common sub-expression elimination, we are actually targeting for expressions that compute the same value.

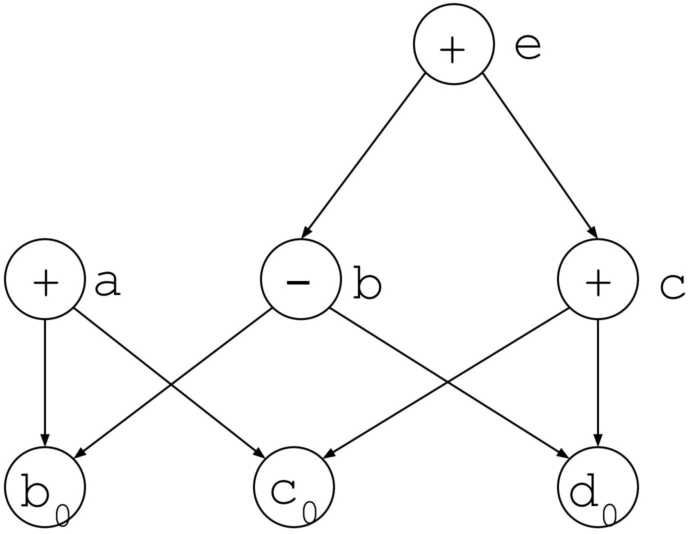
```
a := b + c
b := b - d
c := c + d
e := b + c
```

Common expressions
But do not generate the same result

Optimization of Basic Blocks

- DAG representation identifies expressions that yield the same result

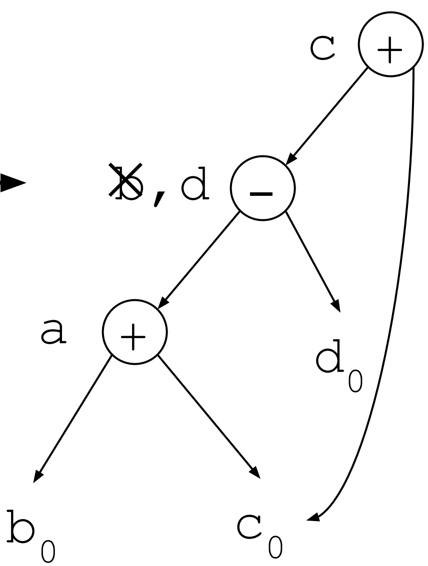
```
a := b + c
b := b − d
c := c + d
e := b + c
```



Optimization of Basic Blocks

- Dead code elimination: Code generation from DAG eliminates dead code.

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




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