Organization

- Introduction
- Classifications of Optimization techniques
- Factors influencing Optimization
- Themes behind Optimization Techniques
- Optimizing Transformations

- Example
- Details of Optimization Techniques

Introduction

- Concerns with machine-independent code optimization
  - 90-10 rule: execution spends 90% time in 10% of the code.
    - It is moderately easy to achieve 90% optimization. The rest 10% is very difficult.
    - Identification of the 10% of the code is not possible for a compiler – it is the job of a profiler.
- In general, loops are the hot-spots

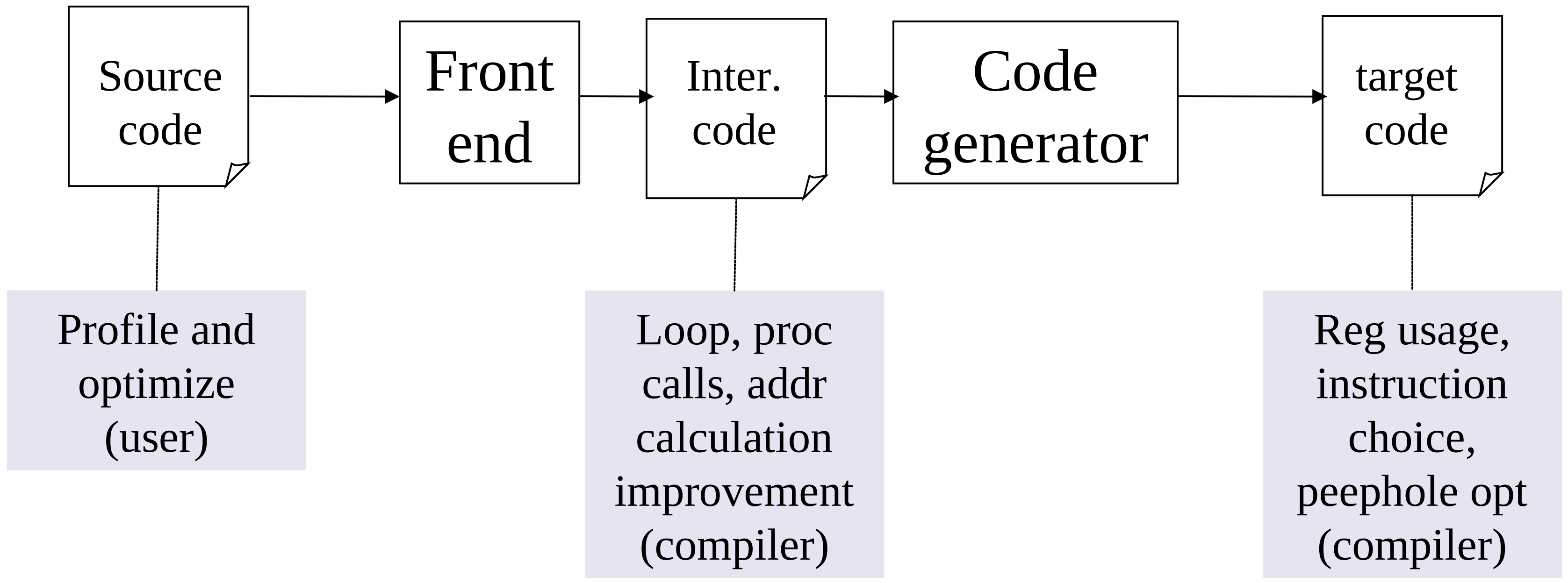
Introduction

- 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

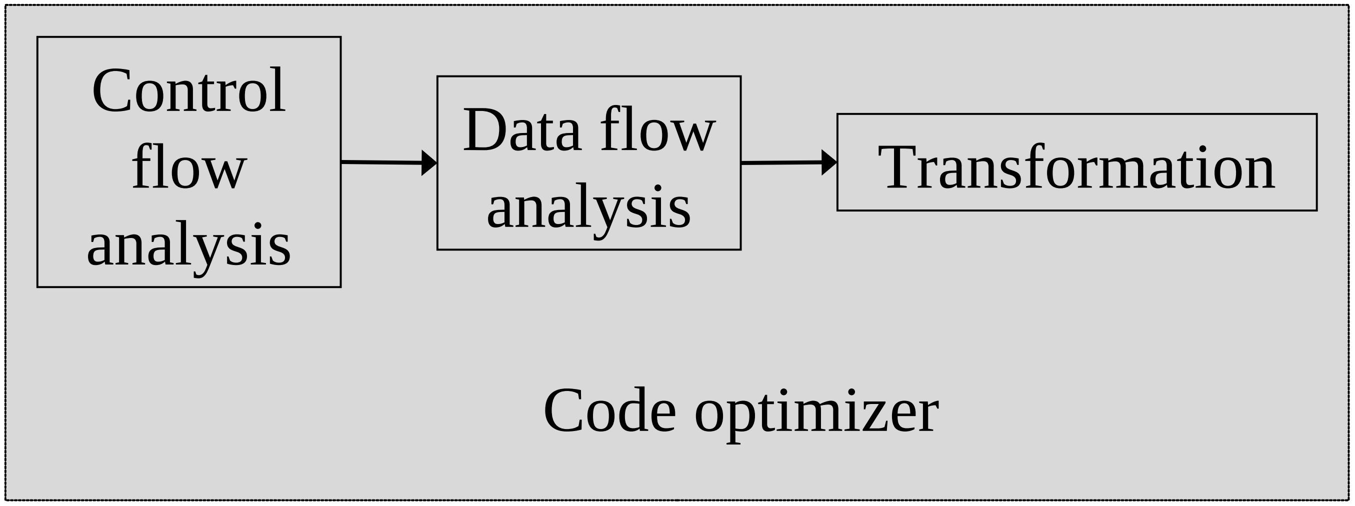
Introduction

- Optimization can be done in almost all phases of compilation.


Source code
Front end
Inter. code
Code generator
target code
Profile and optimize (user)
Loop, proc calls, addr calculation improvement (compiler)
Reg usage, instruction choice, peephole opt (compiler)

Introduction

- Organization of an optimizing compiler



Classifications of Optimization techniques

- Peephole optimization
- Local optimizations
- Global Optimizations
    - Inter-procedural
    - Intra-procedural
- Loop optimization

Factors influencing Optimization

- The target machine: machine dependent factors can be parameterized to compiler for fine tuning
- Architecture of Target CPU:
  - Number of CPU registers
  - RISC vs CISC
  - Pipeline Architecture
  - Number of functional units
- Machine Architecture
  - Cache Size and type
  - Cache/Memory transfer rate

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
- Copy propagation
- Loop optimization
  - Induction variables and strength reduction

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
```

$\downarrow$

```
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
```

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

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                temp := b * c
…              ──────▶          a := temp
…                               …
x := b * c + 5            x := temp + 5
```

Common Subexpression Elimination

- Local common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.

$$\boxed{\begin{array}{l} x = a + b \\ \ldots \\ y = a + b \end{array}} \longrightarrow \boxed{\begin{array}{l} t = a + b \\ x = t \\ \ldots \\ y = t \end{array}}$$

Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

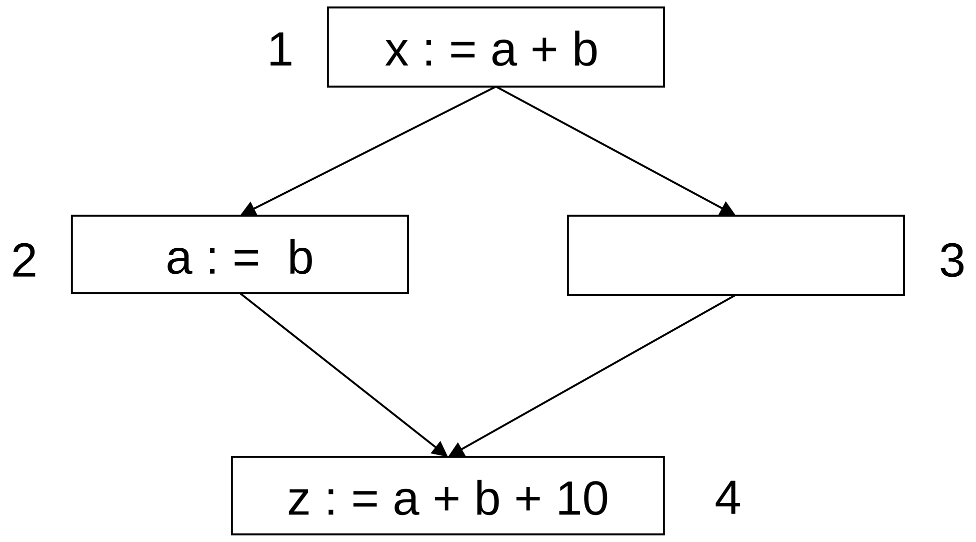
```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

Common Sub-expression Evaluation



"a + b" is not a common sub-expression in 1 and 4

None of the variable involved should be modified in any path

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

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

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

Code Motion

2 *Execution frequency reduction*: reduce execution frequency of partially available expressions (expressions available atleast in one path)

Example:

```
if (a<b) then                  if (a<b) then
      z = x * 2                  temp = x * 2
                                             z = temp
else            ──────►                    else
y = 10                                       y = 10
                                             temp = x * 2
g = x * 2                                  g = temp;
```

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 ) ...
```

Code Motion

- Safety of Code movement

  Movement of an expression *e* from a basic block b_i to another block b_j, is safe if it does not introduce any new occurrence of *e* along any path.

Example: Unsafe code movement

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

Strength Reduction

- Replacement of an operator with a less costly one.

Example:

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

- Typical cases of strength reduction occurs in address calculation of array references.
- Applies to integer expressions involving induction variables (loop optimization)

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

- Examples:

```
DEBUG:=0
if (DEBUG) print
```

$\leftarrow$ Can be eliminated

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.

Local Copy Propagation

- <u>Local</u> copy propagation
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions as you go

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

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

Redundant instruction elimination

- Redundant load/store: see if an obvious replacement is possible

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

  Can eliminate the second instruction without needing any global knowledge of ***a***

- Unreachable code: identify code which will never be executed:

```
#define DEBUG 0
if( DEBUG) {
    print debugging info
}
```

$\longrightarrow$

```
    if (0 != 1) goto L2
    print debugging info

L2:
```

Algebraic identities

- Worth recognizing single instructions with a constant operand:

```
A * 1 = A
A * 0 = 0
A / 1 = A
        A * 2 = A + A
```

  More delicate with floating-point

- Strength reduction:

```
A ^ 2 = A * A
```

Objective

- Why would anyone write **`X * 1`**?
- Why bother to correct such obvious junk code?
- In fact one might write

```
#define MAX_TASKS  1
...
a = b * MAX_TASKS;
```

- Also, seemingly redundant code can be produced by other optimizations. This is an important effect.

Replace Multiply by Shift

- `A := A * 4;`
  - Can be replaced by 2-bit left shift (signed/unsigned)
  - But must worry about overflow if language does
- `A := A / 4;`
  - If unsigned, can replace with shift right
  - But shift right arithmetic is a well-known problem
  - Language may allow it anyway (traditional C)

The right shift problem

- Arithmetic Right shift:
  - shift right and use sign bit to fill most significant bits

    -5 111111...1111111011

    SAR 111111...1111111101

    which is -3, not -2
  - in most languages -5/2 = -2

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:

```
X * 125   =    x * 128 - x*4 + x
```

  - two shifts, one subtract and one add, which may be faster than one multiply
  - Note similarity with efficient exponentiation method

Folding Jumps to Jumps

- A jump to an unconditional jump can copy the target address

```
            JNE lab1
         ...
lab1:       JMP lab2
```

Can be replaced by:

```
        JNE lab2
```

As a result, lab1 may become dead (unreferenced)

Jump to Return

- A jump to a return can be replaced by a return

```
              JMP lab1
            ...
lab1: RET
```

  - Can be replaced by

```
RET
```

lab1 may become dead code

Usage of Machine idioms

- Use machine specific hardware instruction which may be less costly.

i := i + 1

ADD i, #1 $\longrightarrow$ INC i

Local Optimization

Optimization of Basic Blocks

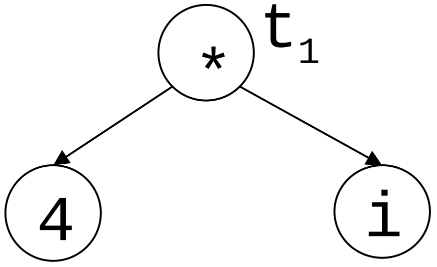
- Many structure preserving transformations can be implemented by construction of DAGs of basic blocks

DAG representation of Basic Block (BB)

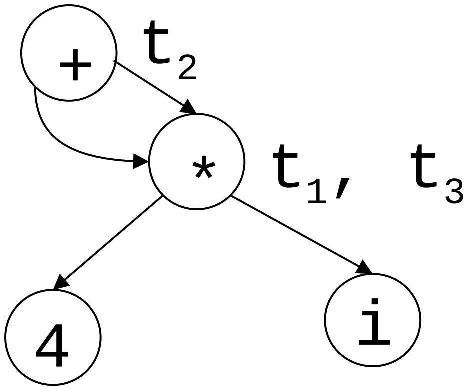
- Leaves are labeled with unique identifier (var name or const)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (identifiers)
- Edges relates operands to the operator (interior nodes are operator)
- Interior node represents computed value
    - Identifier in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

Example: DAG for BB

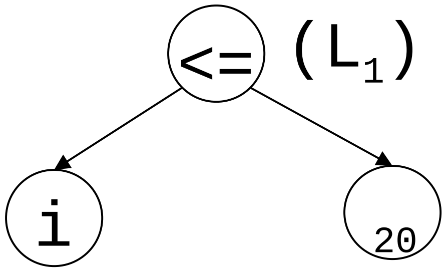
```
t₁ := 4 * i
```



```
t₁ := 4 * i
t₃ := 4 * i
t₂ := t₁ + t₃
```



```
if (i <= 20)goto L₁
```



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)

Construction of DAGs for BB

- Data structure and functions:
  - Node:
    1) Label: label of the node
    2) Left: pointer to the left child node
    3) Right: pointer to the right child node
    4) List: list of additional labels (empty for leaves)
  - **Node (*id*)**: returns the most recent node created for *id*. Else return *undef*
  - **Create(*id,l,r*)**: create a node with label *id* with *l* as left child and *r* as right child. *l* and *r* are optional params.

Construction of DAGs for BB

- Method:

  For each 3AC, *A* in *B*

  A if of the following forms:

  1. $x := y$ op z
  2. $x :=$ op y
  3. $x := y$

```
1. if ((n_y = node(y)) == undef)
       n_y = Create (y);
       if (A == type 1)
         and ((n_z = node(z)) == undef)
        n_z = Create(z);
```

Construction of DAGs for BB

2. If (*A* == type 1)

   Find a node labelled '*op*' with left and right as n_y and n_z respectively [determination of common sub-expression]

   If (not found) n = Create (op, $n_{y,}$ n_z);

   If (*A* == type 2)

   Find a node labelled '*op*' with a single child as n_y

   If (not found) n = Create (op, n_y);

   If (*A* == type 3) n = Node (*y*);

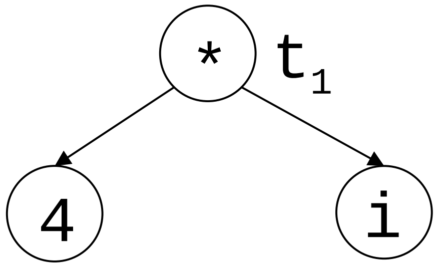
3. Remove x from Node(x).list

   Add *x* in n.list

   Node(*x*) = n;

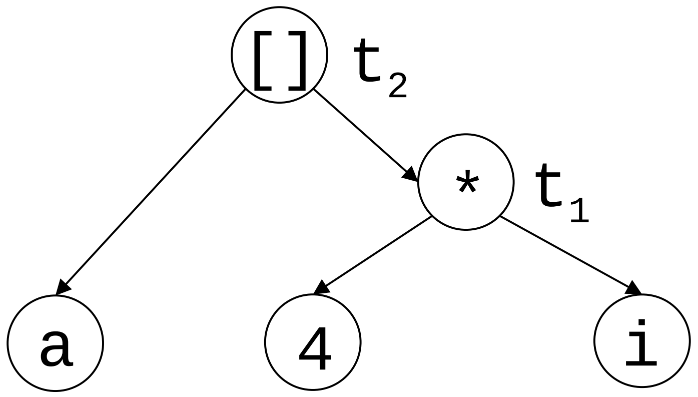
Example: DAG construction from BB

t_1 := 4 * i



Example: DAG construction from BB

```
t₁ := 4 * i
t₂ := a [ t₁ ]
```

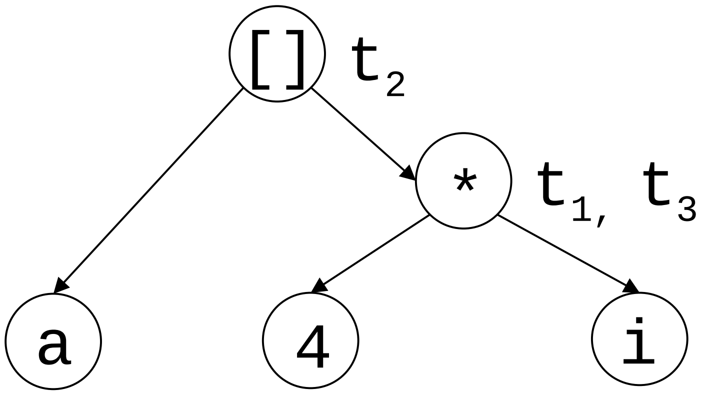


Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i



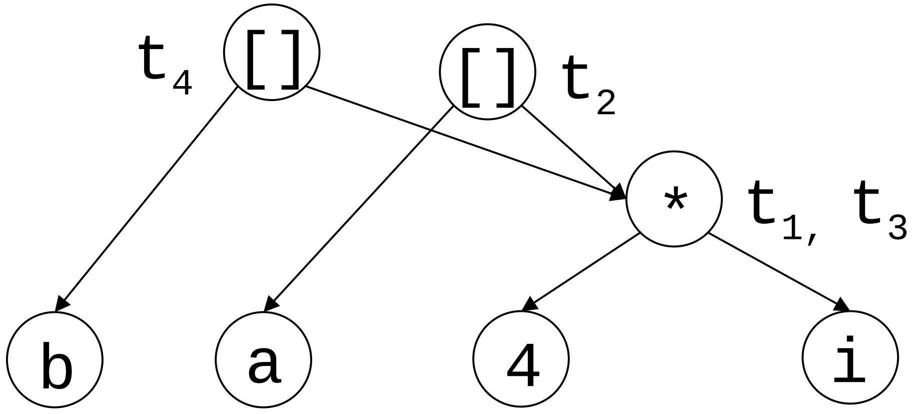
Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]



Example: DAG construction from BB

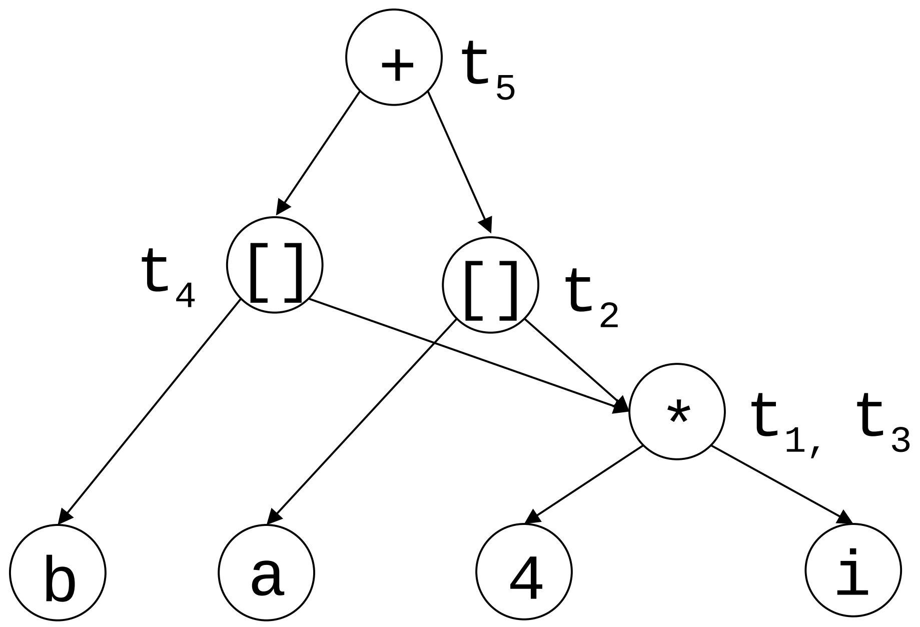
t_1 := 4 * i

t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

t_5 := t_2 + t_4



Example: DAG construction from BB

t_1 := 4 * i

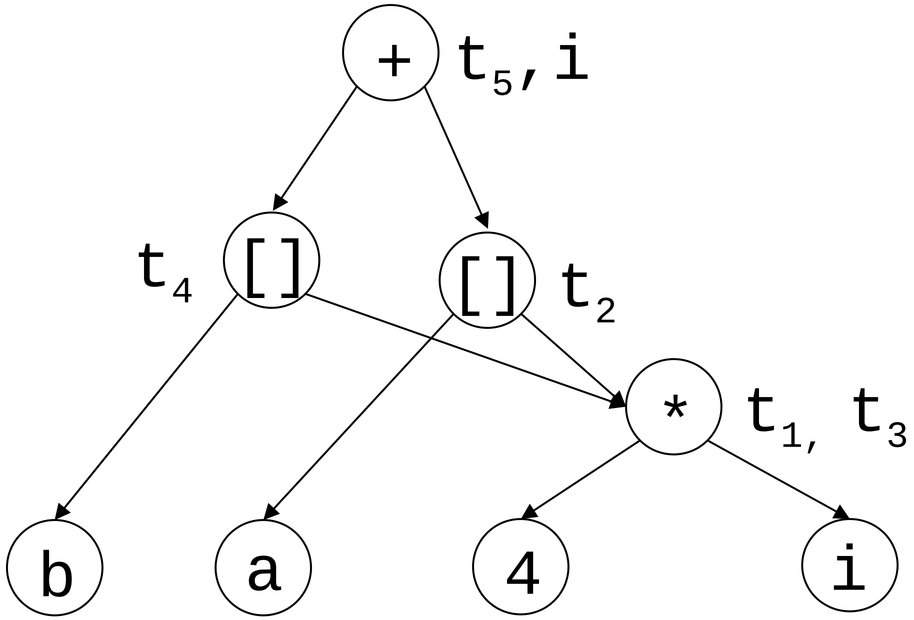
t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

t_5 := t_2 + t_4

i := t_5



DAG of a Basic Block

- Observations:
  - A leaf node for the initial value of an id
  - A node *n* for each statement *s*
  - The children of node *n* are the last definition (prior to *s*) of the operands of *n*

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
```

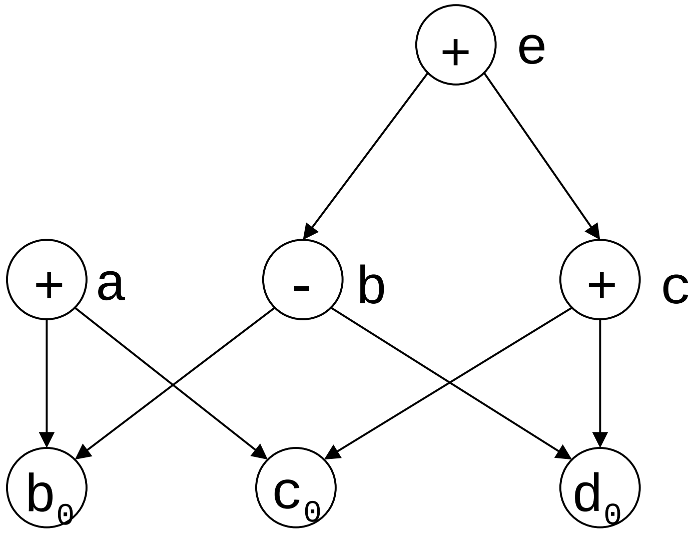
Common expressions

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
```



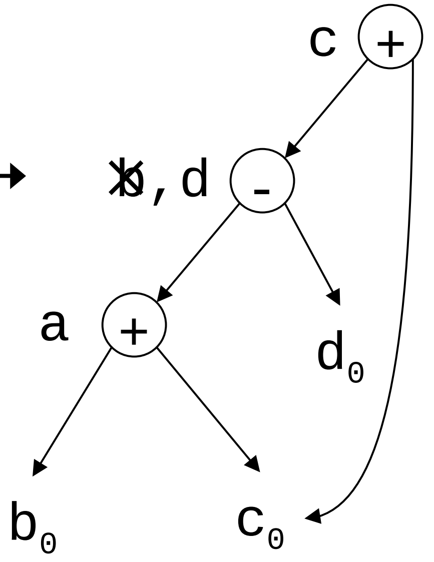
Optimization of Basic Blocks

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

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

b is not live

$\longrightarrow$



$\longrightarrow$

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

Loop Optimization

Loop Optimizations

- Most important set of optimizations
  - Programs are likely to spend more time in loops
- Presumption: Loop has been identified
- Optimizations:
  - Loop invariant code removal
  - Induction variable strength reduction
  - Induction variable reduction

Loops in Flow Graph

- Dominators:

  A node *d* of a flow graph *G* dominates a node *n*, if every path in *G* from the initial node to *n* goes through *d*.

  Represented as: *d dom n*

  Corollaries:

  Every node dominates itself.

  The initial node dominates all nodes in *G.*

  The entry node of a loop dominates all nodes in the loop.

Loops in Flow Graph

- Each node *n* has a unique *immediate dominator m*, which is the last dominator of *n* on any path in *G* from the initial node to *n*.

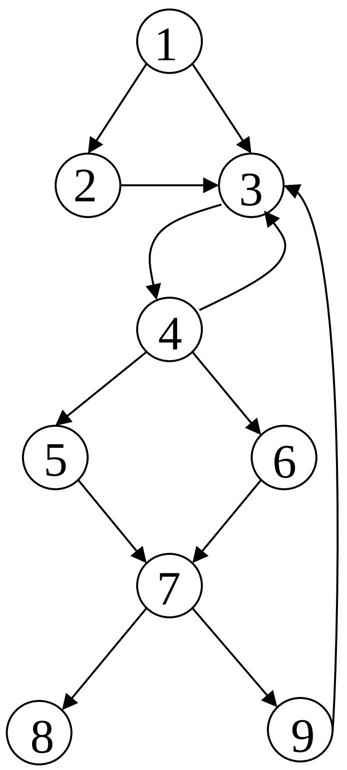
  (d ≠ n) && (d dom n) → d dom m

- Dominator tree (*T*):

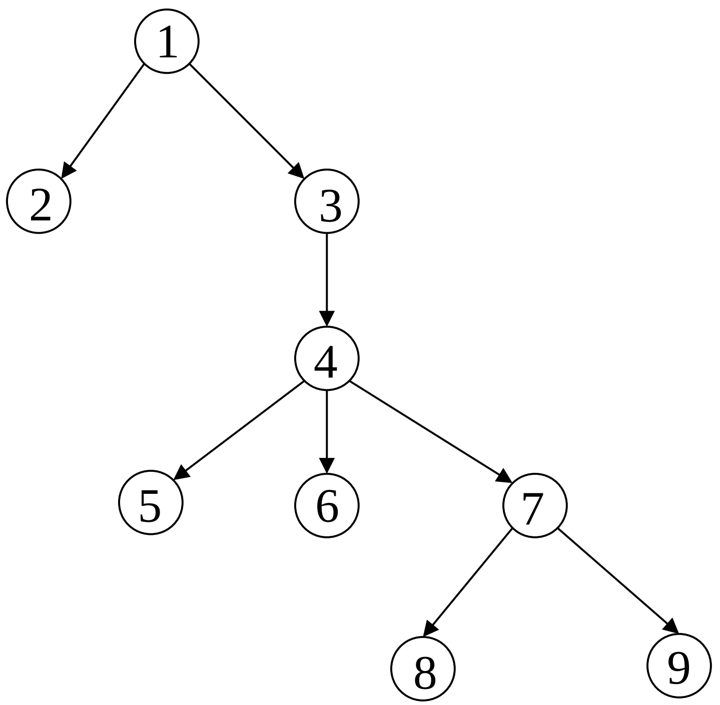
  A representation of dominator information of flow graph *G*.

  - The root node of *T* is the initial node of *G*
  - A node *d* in *T* dominates all node in its sub-tree

Example: Loops in Flow Graph



Flow Graph



Dominator Tree

Loops in Flow Graph

- Natural loops:
  1. A loop has a single entry point, called the “header”. Header dominates all node in the loop
  2. There is at least one path back to the header from the loop nodes (i.e. there is at least one way to iterate the loop)

- Natural loops can be detected by *back edges*.
  - B*ack edges*: edges where the sink node (head) dominates the source node (tail) in *G*

Natural loop construction

- Construction of natural loop for a back edge

  Input: A flow graph G,
  A back edge $n \rightarrow d$

  Output: The set *loop* consisting of all nodes in
  the natural loop of $n \rightarrow d$

  Method:

```
stack := ϵ ;  loop := {d};
insert(n);
while (stack not empty)
m := stack.pop();
for each predecessor p of m do
insert(p)
```

Function: insert (m)

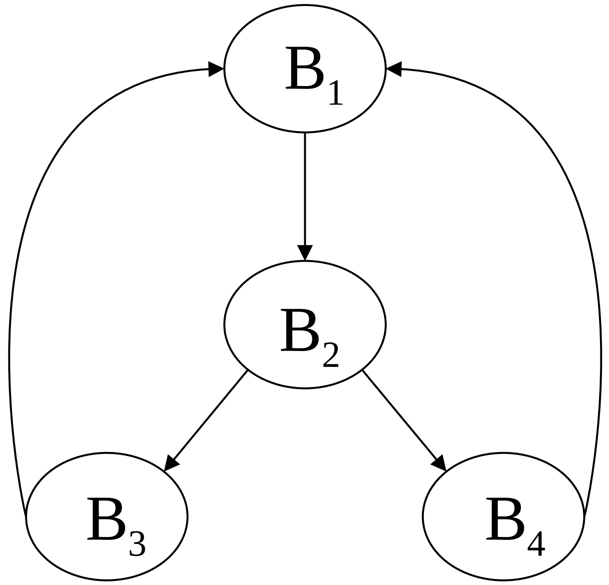
```
if !(m ϵ loop)
        loop := loop U {m}
        stack.push(m)
```

Inner loops

- Property of natural loops:
  - If two loops l_1 and l_2, do not have the same header,
    - l_1 and l_2 are disjoint.
    - One is an inner loop of the other.
- Inner loop: loop that contains no other loop.
    - Loops which do not have the same header.

Inner loops

- Loops having the same header:



It is difficult to conclude which one of $\{B_1, B_2, B_3\}$ and $\{B_1, B_2, B_4\}$ is the inner Loop without detailed analysis of code.

<u>Assumption:</u>
When two loops have the same header they are treated as a single Loop.

Loop Optimization

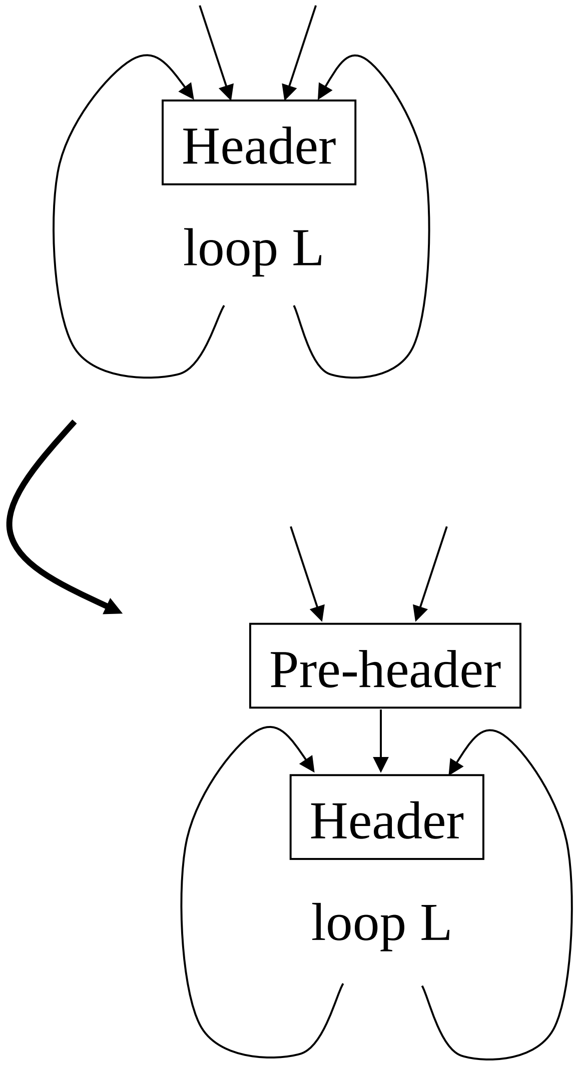
- **Loop interchange**: exchange inner loops with outer loops
- **Loop splitting**: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range.
  - A useful special case is ***loop peeling*** - simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

Loop Optimization

- **Loop fusion**: two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data
- **Loop fission**: break a loop into multiple loops over the same index range but each taking only a part of the loop's body.
- **Loop unrolling**: duplicates the body of the loop multiple times

Loop Optimization

- Pre-Header:
  - Targeted to hold statements that are moved out of the loop
  - A basic block which has only the header as successor
  - Control flow that used to enter the loop from outside the loop, through the header, enters the loop from the pre-header



Loop Invariant Code Removal

- Move out to pre-header the statements whose source operands do not change within the loop.
  - Be careful with the memory operations
  - Be careful with statements which are executed in some of the iterations

Loop Invariant Code Removal

- Rules: A statement S: *x*:=*y* op *z* is loop invariant:
  - *y* and *z* not modified in loop body
  - S is the only statement to modify *x*
  - For all uses of *x*, *x* is in the available def set.
  - For all exit edge from the loop, S is in the available def set of the edges.
  - If S is a load or store (mem ops), then there is no writes to address(*x*) in the loop.

Loop Invariant Code Removal

- Loop invariant code removal can be done without available definition information.

Rules that need change:

- For all use of *x* is in the available definition set
- For all exit edges, if *x* is live on the exit edges, is in the available definition set on the exit edges

- Approx of First rule:
  - *d* dominates all uses of *x*
- Approx of Second rule
  - *d* dominates all exit basic blocks where *x* is live

Loop Induction Variable

- Induction variables are variables such that every time they change value, they are incremented or decremented.
  - Basic induction variable: induction variable whose only assignments within a loop are of the form:

    `i = i +/- C`, where C is a constant.
  - Primary induction variable: basic induction variable that controls the loop execution

    `(for i=0; i<100; i++)`

    `i` (register holding i) is the primary induction variable.
  - Derived induction variable: variable that is a linear function of a basic induction variable.

Loop Induction Variable

- Basic: r4, r7, r1
- Primary: r1
- Derived: r2

```
       r1 = 0
       r7 = &A

Loop:  r2 = r1 * 4
       r4 = r7 + 3
       r7 = r7 + 1
       r10 = *r2
       r3 = *r4
       r9 = r1 * r3
       r10 = r9 >> 4
       *r2 = r10
       r1 = r1 + 4
       If(r1 < 100) goto Loop
```

Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables.
- Rules: (S: *x* := *y* op *z*)
  - *op* is *, <<, +, or –
  - *y* is a induction variable
  - *z* is invariant
  - No other statement modifies *x*
  - *x* is not *y* or *z*
  - *x* is a register

Induction Variable Strength Reduction

- Transformation:

  Insert the following into the bottom of pre-header:

  new_reg = expression of target statement S

  if (opcode(S)) is not add/sub, insert to the bottom of the preheader

  new_inc = inc(y,op,z)

  else

  new_inc = inc(x)

  Insert the following at each update of *y*

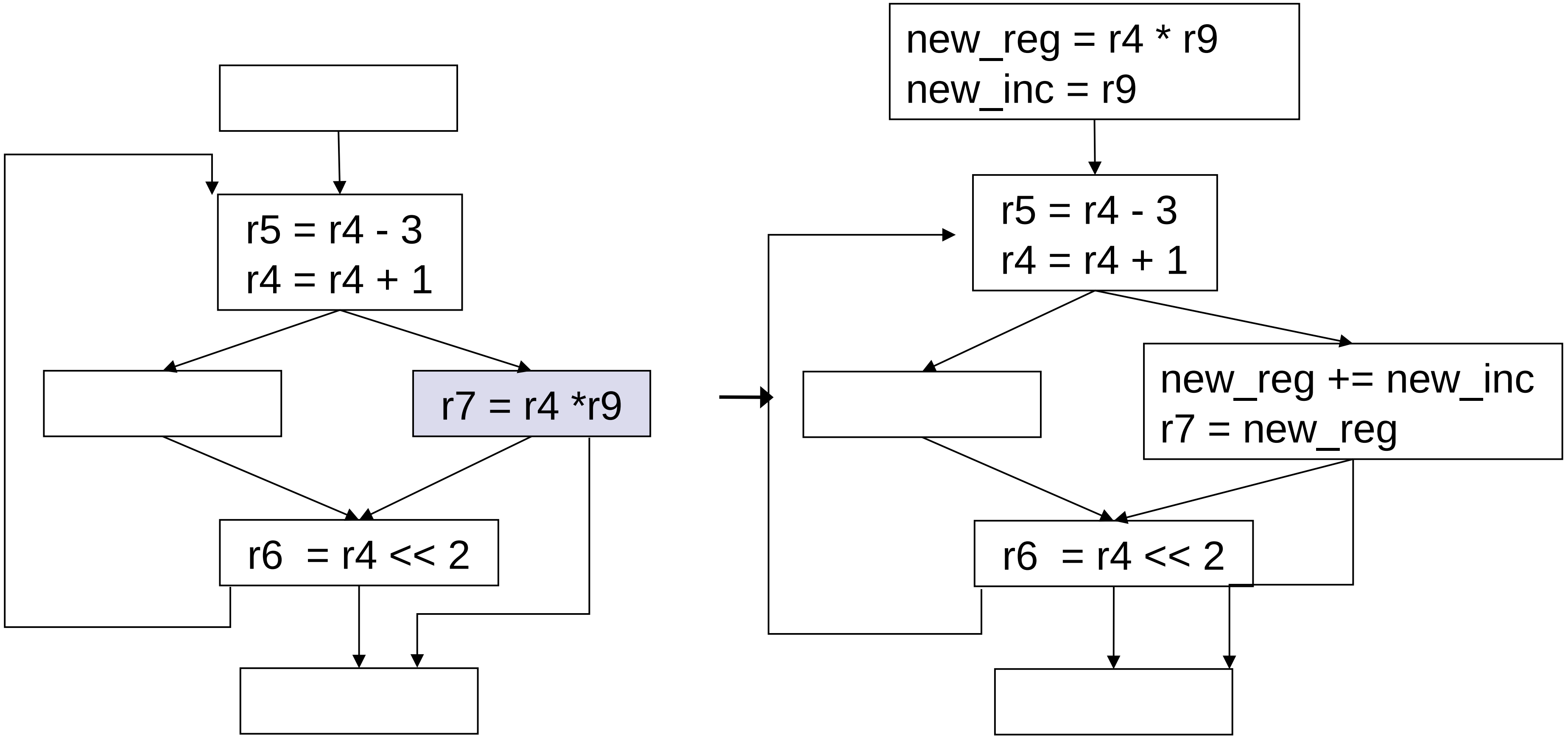
  new_reg = new_reg + new_inc

  Change S: *x = new_reg*

Function: inc()

Calculate the amount of inc for 1st param.

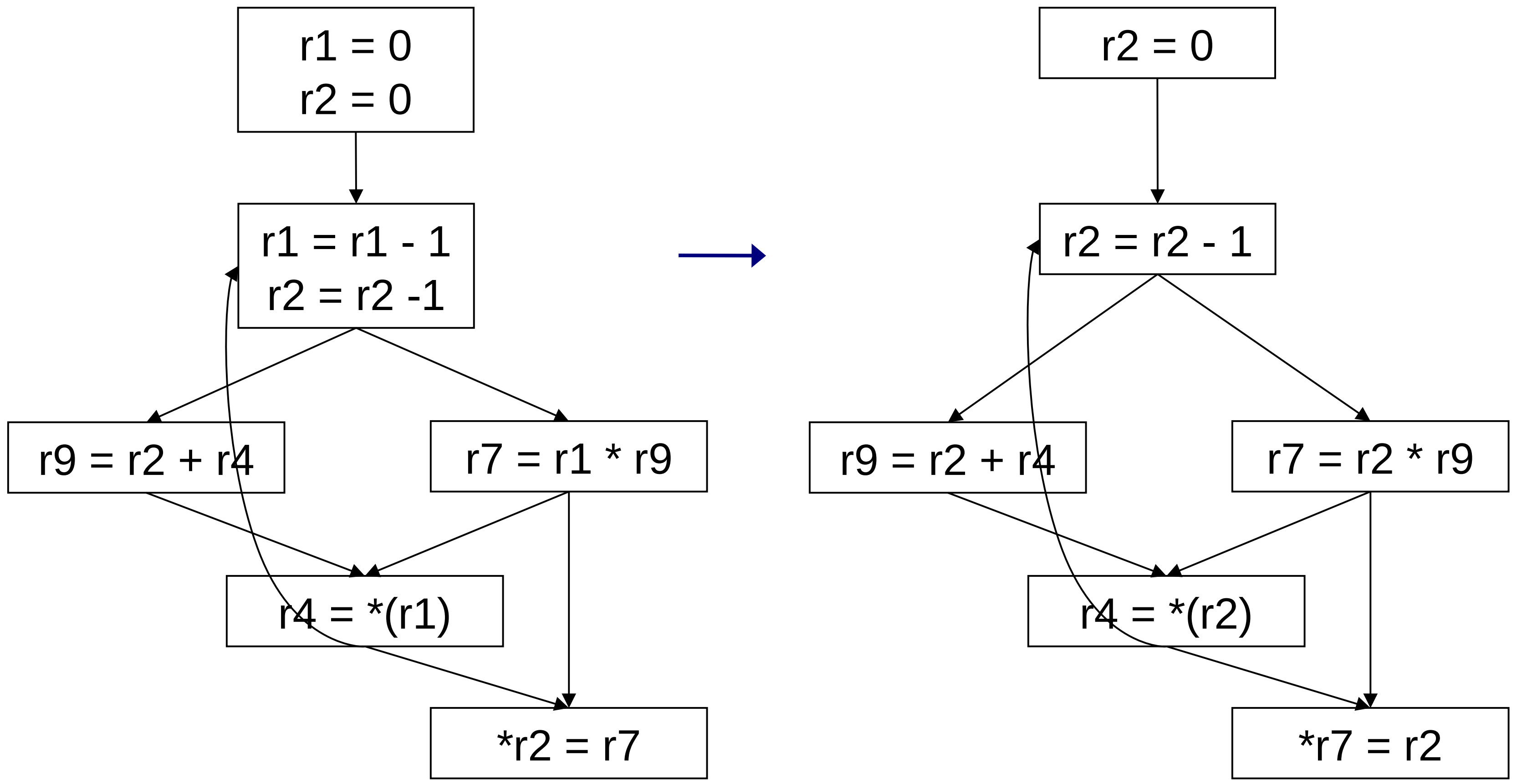
Example: Induction Variable Strength Reduction



Induction Variable Elimination

- Remove unnecessary basic induction variables from the loop by substituting uses with another basic induction variable.
- Rules:
  - Find two basic induction variables, *x* and *y*
  - *x* and *y* in the same family
    - Incremented at the same place
  - Increments are equal
  - Initial values are equal
  - *x* is not live at exit of loop
  - For each BB where *x* is defined, there is no use of x between the first and the last definition of *y*

Example: Induction Variable Elimination



Induction Variable Elimination

- Variants:

Complexity of elimination

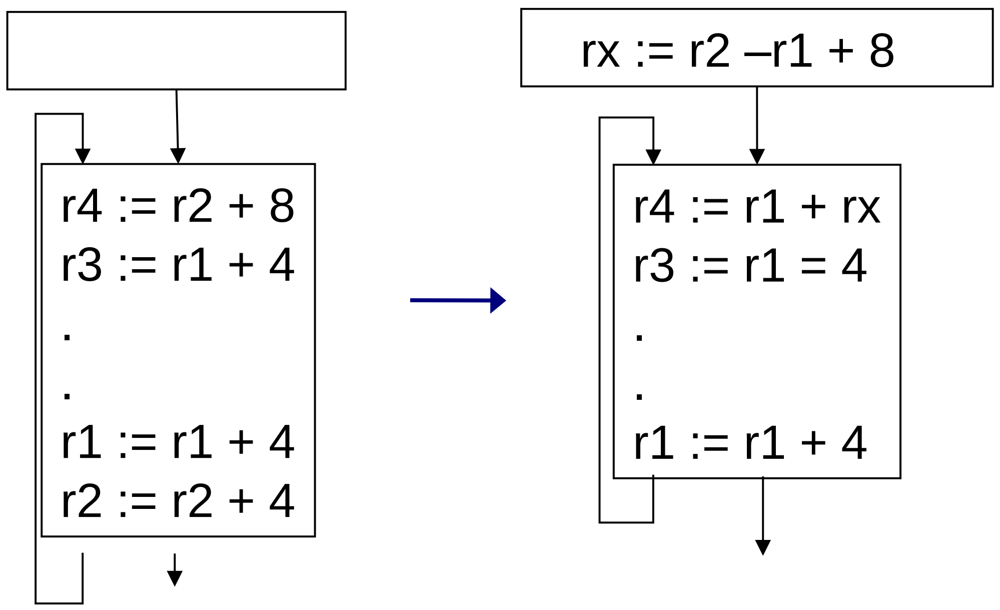
1. Trivial: induction variable that are never used except to increment themselves and not live at the exit of loop
2. Same increment, same initial value (discussed)
3. Same increment, initial values are a known constant offset from one another
4. Same increment, nothing known about the relation of initial value
5. Different increments, nothing known about the relation of initial value

- 1,2 are basically free
- 3-5 require complex pre-header operations

Example: Induction Variable Elimination

- Case 4: Same increment, unknown initial value

  For the induction variable that we are eliminating, look at each non-incremental use, generate the same sequence of values as before. If that can be done without adding any extra statements in the loop body, then the transformation can be done.



Loop Unrolling

- Replicate the body of a loop (N-1) times, resulting in total N copies.
  - Enable overlap of operations from different iterations
  - Increase potential of instruction level parallelism (ILP)
- Variants:
  - Unroll multiple of known trip counts
  - Unroll with remainder loop
  - While loop unroll

Constant Folding

- Simplify operation based on values of target operand
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about FP?
  - Evaluate conditional branch, replace with BRU or noop
    - if (1 < 2) goto BB2 → goto BB2
    - if (1 > 2) goto BB2 → convert to a noop **Dead code**
- Algebraic identities
  - r1 = r2 + 0, r2 – 0, r2 | 0, r2 ^ 0, r2 << 0, r2 >> 0 → r1 = r2
  - r1 = 0 * r2, 0 / r2, 0 & r2 → r1 = 0
  - r1 = r2 * 1, r2 / 1 → r1 = r2

Strength Reduction

- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Mult by power of 2: r1 = r2 * 8 → r1 = r2 << 3
  - Div by power of 2: r1 = r2 / 4 → r1 = r2 >> 2
  - Rem by power of 2: r1 = r2 % 16 → r1 = r2 & 15
- More exotic
  - Replace multiply by constant by sequence of shift and adds/subs
    - r1 = r2 * 6
      - r100 = r2 << 2; r101 = r2 << 1; r1 = r100 + r101
    - r1 = r2 * 7
      - r100 = r2 << 3; r1 = r100 – r2

Dead Code Elimination

- Remove statement d: *x* := *y* op *z* whose result is never consumed.
- Rules:
  - DU chain for d is empty
  - *y* and *z* are not live at d

Constant Propagation

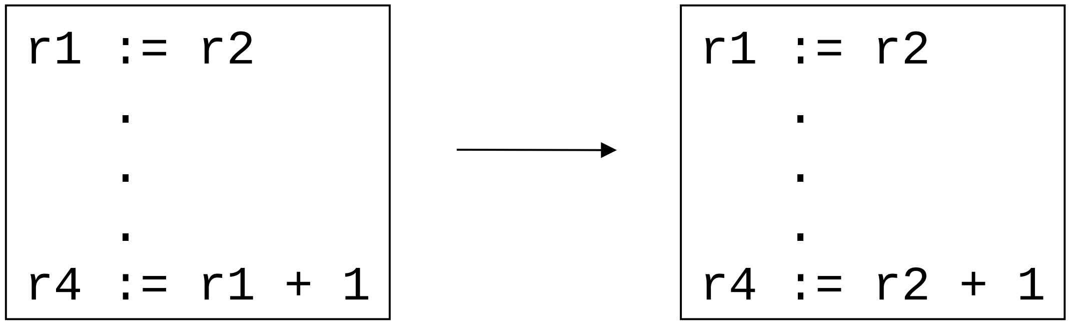
- Forward propagation of moves/assignment of the form

  d: `rx := L` where `L` is literal

  - Replacement of "rx" with "L" wherever possible.
  - d must be available at point of replacement.

Forward Copy Propagation

- Forward propagation of RHS of assignment or mov's.



```
r1 := r2
    .
    .
    .
r4 := r1 + 1
```

```
r1 := r2
    .
    .
    .
r4 := r2 + 1
```

  - Reduce chain of dependency
  - Possibly create dead code

Forward Copy Propagation

- Rules:

  Statement d_S is source of copy propagation

  Statement d_T is target of copy propagation

  - d_S is a mov statement
  - src(d_S) is a register
  - d_T uses dest(d_S)
  - d_S is available definition at d_T
  - src(d_S) is a available expression at d_T

Backward Copy Propagation

- Backward propagation of LHS of an assignment.

```
dT: r1 := r2 + r3     → r4 := r2 + r3
    r5 := r1 + r6     → r5 := r4 + r6
dS: r4 := r1          → Dead Code
```

- Rules:
  - d_T and d_S are in the same basic block
  - dest(d_T) is register
  - dest(d_T) is not live in *out*[B]
  - dest(d_S) is a register
  - d_S uses dest(d_T)
  - dest(d_S) not used between d_T and d_S
  - dest(d_S) not defined between d_1 and d_S
  - There is no use of dest(d_T) after the first definition of dest(d_S)

Local Common Sub-Expression Elimination

- Benefits:
  - Reduced computation
  - Generates mov statements, which can get copy propagated
- Rules:
  - d_S and d_T has the same expression
  - src(d_S) == src(d_T) for all sources
  - For all sources *x*, *x* is not redefined between d_S and d_T

d_S: r1 := r2 + r3

d_T: r4 := r2 + r3

d_S: r1 := r2 + r3
r100 := r1

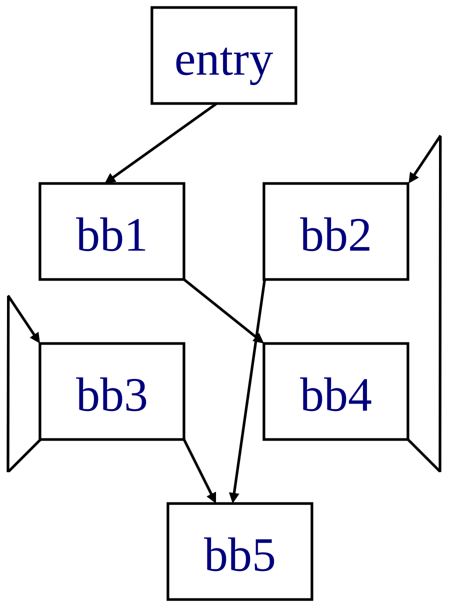
d_T: r4 := r100

Global Common Sub-Expression Elimination

- Rules:
  - d_S and d_T has the same expression
  - src(d_S) == src(d_T) for all sources of d_S and d_T
  - Expression of d_S is available at d_T

Unreachable Code Elimination

```
Mark initial BB visited
to_visit = initial BB
while (to_visit not empty)
    current = to_visit.pop()
    for each successor block of current
            Mark successor as visited;
            to_visit += successor
    endfor
endwhile
Eliminate all unvisited blocks
```



Which BB(s) can be deleted?