PURDUE
UNIVERSITY

Department of Computer Sciences

Redundancy Removal

CS502

Source of Redundant Operations

- Redundant operations are mainly due to
  - Low level operations exposed after high-level program statements are translated
  - A comprehensive program being “specialized” (as the result of #define, e.g.)
  - Just-in-time optimization (for mobile code) exposes redundancies in a particular program segment (when some variables have values fixed).

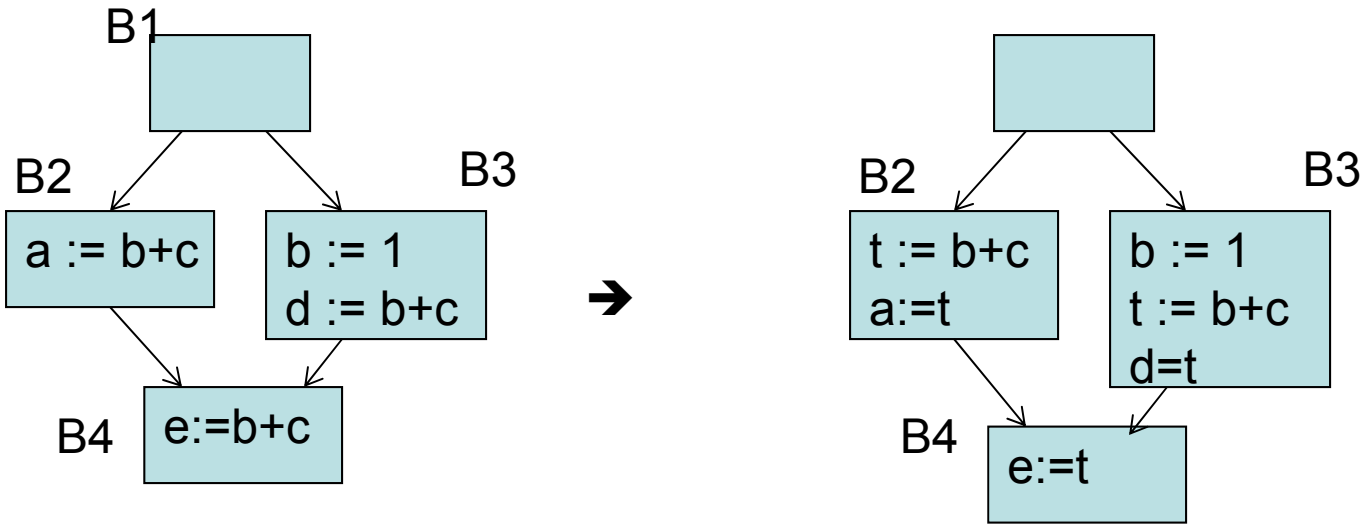
Two main kinds of redundancy

- Common sub-expressions (CSE)
- Loop invariant expressions
- Constant expressions
  - Constant expressions may lead to the discoveries of “dead” branches → unreachable code
  - There is a cycle between dead-code discoveries and constant discoveries.

CSE Removal Based on Available Expressions Analysis

- Suppose an operation *f* is performed at program point *p* to evaluate an expression *E.* If we find out that no matter how the program execution reaches *p*, the value of *E* will have already been computed early on by another operation, then the operation f is redundant.
  - We could simply copy the value computed early on
- It is important that the value of E is always *available*.
- We will look at an example.

A Simple Example

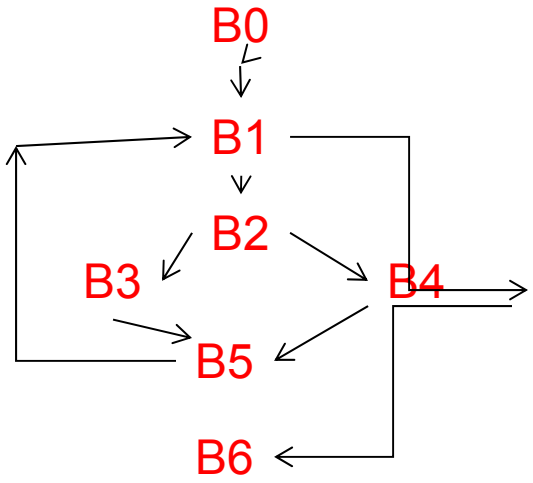


"Deep" Analysis of CSE

- Conventional available expression analysis only recognizes expressions that have the identical textual form.
- There may be expressions that always have identical values (if we interpret the symbolic values correctly) but may not have identical textual forms
  - Such expressions are called *congruent expressions*
  - Recognizing congruent expressions will expose more available expressions

int B[size][size], A[size][size]

\\Loop body is:
If readB
  v = B[row,j] ;
Else
  v = A[row,j] ;
C[row, j] = v;



B0
```
j = 0;
```
B1
```
L1: if j >= 100 goto L2
```
B2
```
if (readB) goto L3
```
B3
```
t0 = row * size
t1= t0 + j
t2 = t1 * 4
t3 = baseA + t2
v = mem[t3]
goto L4
```
```
L3: t4 = row * size
t5 = t4 + j
t6 = t5 * 4
t7 = baseB + t6
v = mem[t7]
```
B4
```
L4: t8 = row * size
t9  = t8 + j
t10 = t9 * 4
t11 = baseC + t10
mem[t11] = v
j = j + 1
goto L1
```
B5
```
L2:
```
B6

row * size
is textually identical in B3, B4 and B5.

t0+j, t4+j, t8+j are congruent.

t1*4, t5*4, t9*4 are congruent.

Conventional Analysis of Available Expressions (w/o recognizing congruent expressions)

- To simplify our notation, assume the expressions we consider are in the form of $x \oplus y$
- An expression $x \oplus y$ is *available* at a node *n* in the flow graph if, on every path from the entry node of the graph to node *n,* $x \oplus y$ (or one of its congruent expressions) is computed at least once *and* there are **no definitions** of *x* or *y* since the most recent occurrence of $x \oplus y$ on that path.
- We want to compute AEin(B) and AEout(B) for each basic block B, which represent the sets of expression available on entry and exit of B, respectively.
  - AEin and AEout are sets of expressions in the form of $x \oplus y$

We first go through all basic blocks. For each block B, we establish two pieces of invariant information, *EVAL(B)* and *KILL(B)*:

- EVAL(B) is the set of expressions in the form of $x \oplus y$ *within* basic block *B* such that neither x nor y is modified between the operation of $x \oplus y$ and the exit of B.

- KILL(B) is the set of expressions outside B which become unavailable after execution of B because at least one of the operands gets modified in B.

In other words, any node that computes $x \oplus y$ *generates* $\{x \oplus y\}$, and any definition of *x* or *y kills* $\{x \oplus y\}$;

We have the following flow equations:

$$AEin(B) = \bigcap_{j \in Pred(B)} AEout(j)$$

$$AEout(B) = EVAL(B) \cup (AEin(B) - KILL(B))$$

Before we start iterating, AEout(entry) is initialized as an empty set and AEout(B) is initialized to "all expression" for all other nodes *B. (Why?)*

The initial worklist contains successors of the entry if we use the worklist approach for iteration.

Common sub-expression Elimination

Given a flow-graph statement $s : t \leftarrow x \oplus y$, where the expression $x \oplus y$ is *available* at s, the computation within s can be eliminated.

Algorithm. Compute *reaching expressions*, that is, find statements of the form $n : v \leftarrow x \oplus y$, such that the path from n to s does not compute $x \oplus y$ or define x or y.

Choose a new temporary w, and for such n, rewrite as

$n : w \leftarrow x \oplus y$

$n' : v \leftarrow w$

Finally, modify statement s to be

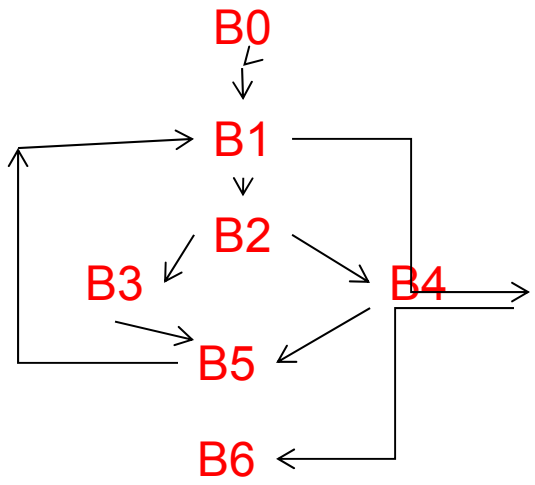
$s : t \leftarrow w$

We will rely on a later compiler pass called “copy propagation” to remove some or all of the extra assignment quadruples.

Recognizing Congruent Expressions

- Obviously textually identical expressions are not necessarily congruent
  - That is why we may kill an available expression when one of the operands are modified
  - To recognize congruent expressions, we assign each definition of variable with “signature”
  - D0: a +b has the signature (+,a,b)
  - The defs of the same signature will have the same value number
  - In general a signature is of the form (op, sig1, sig2)
  - In (+,a,b) above, a and b are assumed to have the “input” values or the initial values of a and b respectively

B0
```
j = 0;
```
B1
```
L1: if j >= 100 goto L2
```
```
if (readB) goto L3
```
B2
```
t0 = row * size
t1= t0 + j
t2 = t1 * 4
t3 = baseA + t2
v = mem[t3]
goto L4
```
B3
```
L3: t4 = row * size
t5 = t4 + j
t6 = t5 * 4
t7 = baseB + t6
v = mem[t7]
```
B4
```
L4: t8 = row * size
t9  = t8 + j
t10 = t9 * 4
t11 = baseC + t10
mem[t11] = v
j = j + 1
goto L1
```
B5
```
L2:
```
B6



t0's signature (*, row, size) is assigned a value number, say e1. Row is assumed to be of the initial value, size is a constant.

t4 and t8 have the same value number e1.

t1, t5 and t9 have the value number e2, with signature (+,e1,j). (What is j?)

t2, t6, and t10 have the value number e3, with signature (*, e2, 4).

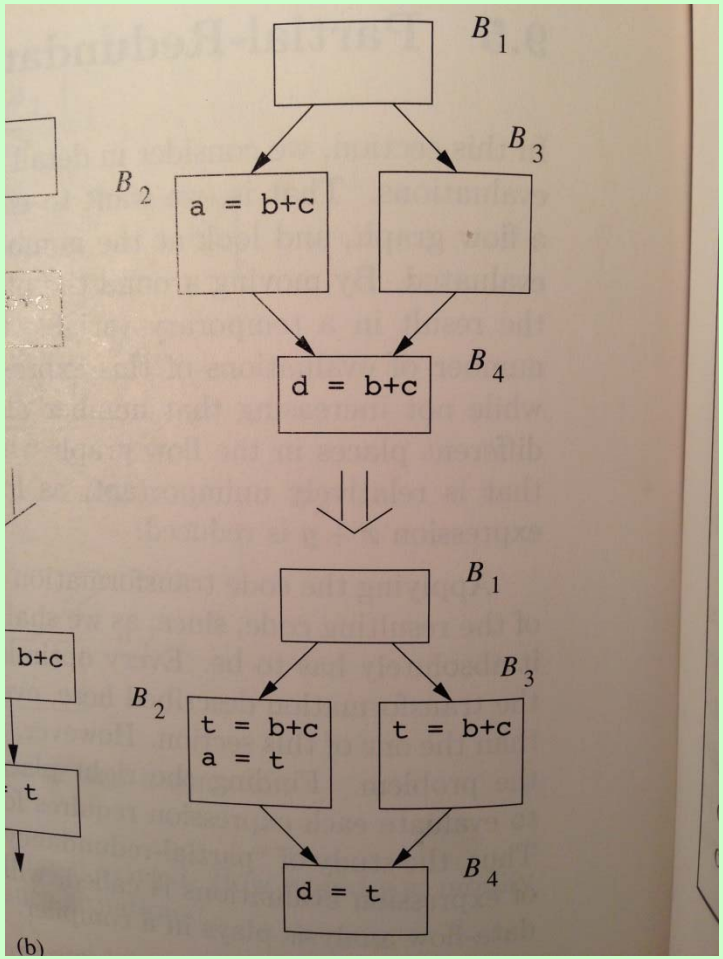
We can now propagate AV in the form of signatures. Intersect, but no need to kill the expression.

How to recognize congruent expressions?

- Or, equivalently, how do we form signatures?
- This concerns transforming the program into a static single assignment (SSA) form, such that every use of a variable has a unique reaching definition.
- SSA is critical to symbolic analysis of a program
- We discuss SSA in a later lecture

Partial redundancy removal

- A redundant expression might not be available along every path reaching the optimization target in the control flow
- Insert the evaluation of the expression in the exposed path



PURDUE UNIVERSITY

- Avoid introducing new redundancy due to control flow not reaching the optimization path

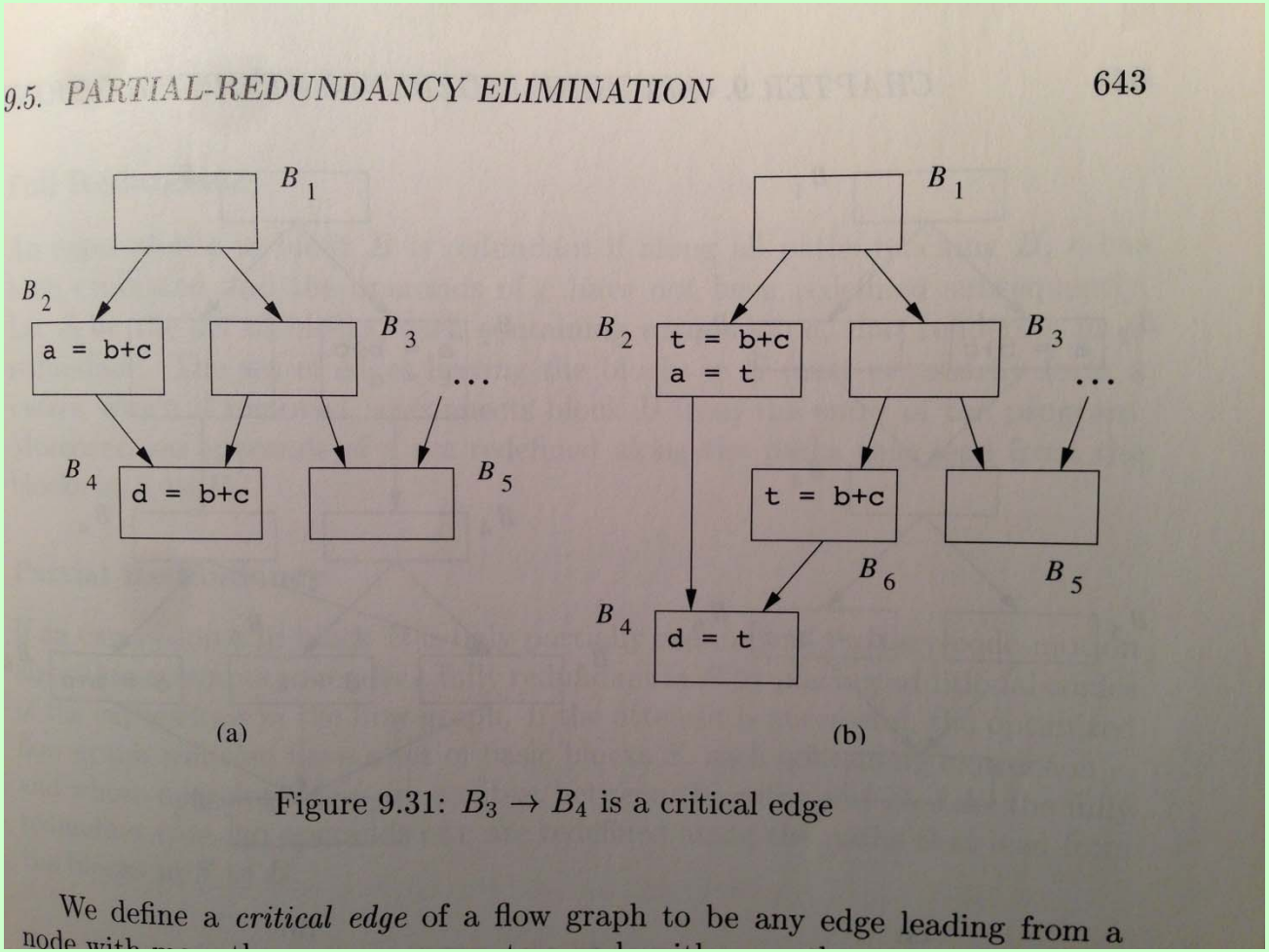


Figure 9.31: $B_3 \rightarrow B_4$ is a critical edge

We define a *critical edge* of a flow graph to be any edge leading from a node with

PURDUE UNIVERSITY

- There are difficult cases in which, to avoid introducing new redundancy, branches may be duplicated to ensure correct def-use chains

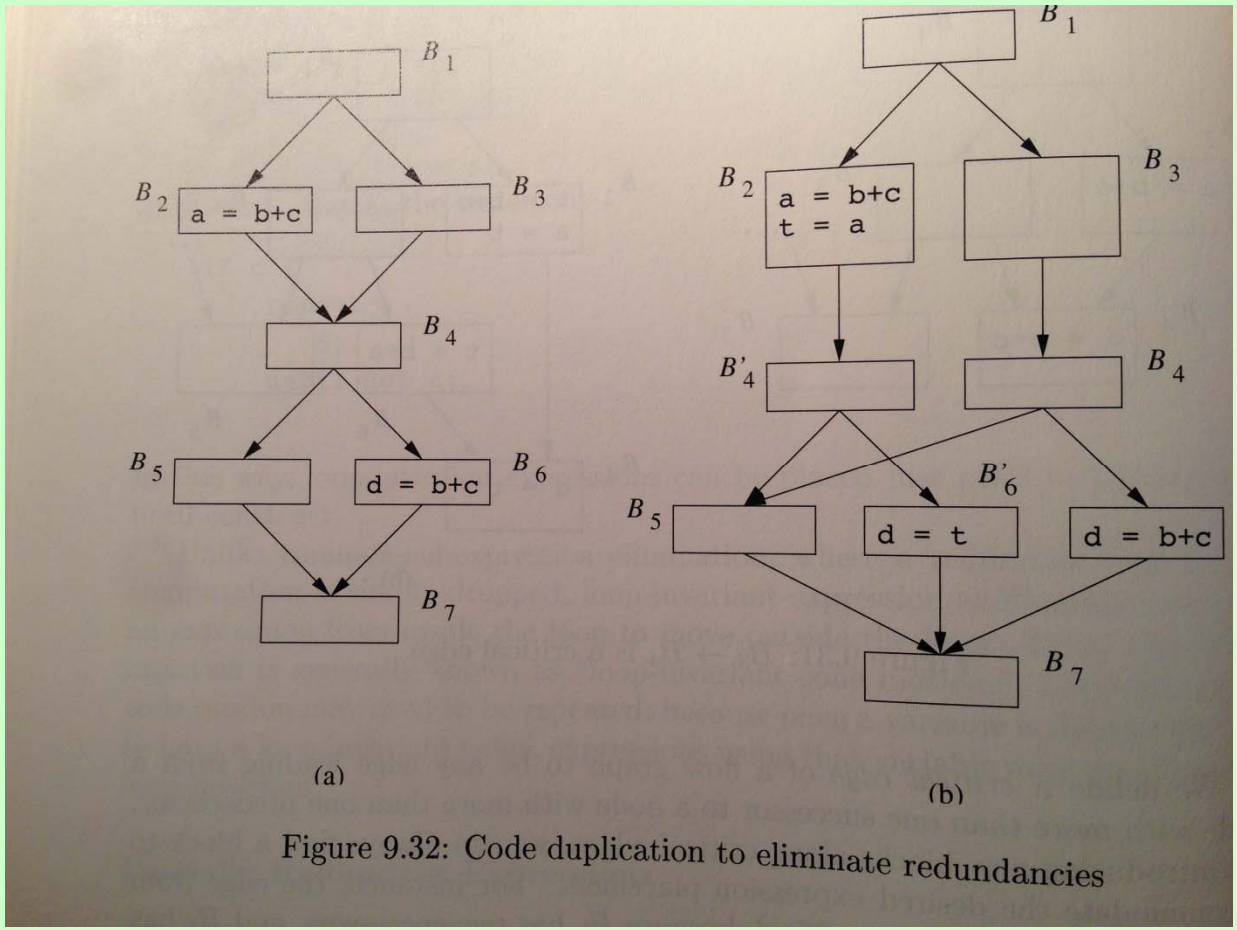


Figure 9.32: Code duplication to eliminate redundancies

PURDUE UNIVERSITY

Constant Folding

- Example:
- #define chunksize 20
- #define buffersize 1000
- for (i=0; i<1000; i++) {

  x = chunksize * i;

  p = malloc(x*buffersize*size(node)); }

PURDUE
UNIVERSITY

Loop invariant code motion

- If a loop contains a statement $t \leftarrow a \oplus b$ such that a has the same value each time around the loop, and b has the same value each time, then t will also have the same value each time. We would like to *hoist* the computation out of the loop, so it is computed just once instead of every time.
- Example:

```
while (p!=NULL) {
    i++;
    compare(a, list.message[i].val);
    p = list.message[i].sym_ptr;  }
```

At the source level, one does not see redundant operation repeating in this loop

But at 3AC level, to calculate the address of list.message[i].val, the operations are

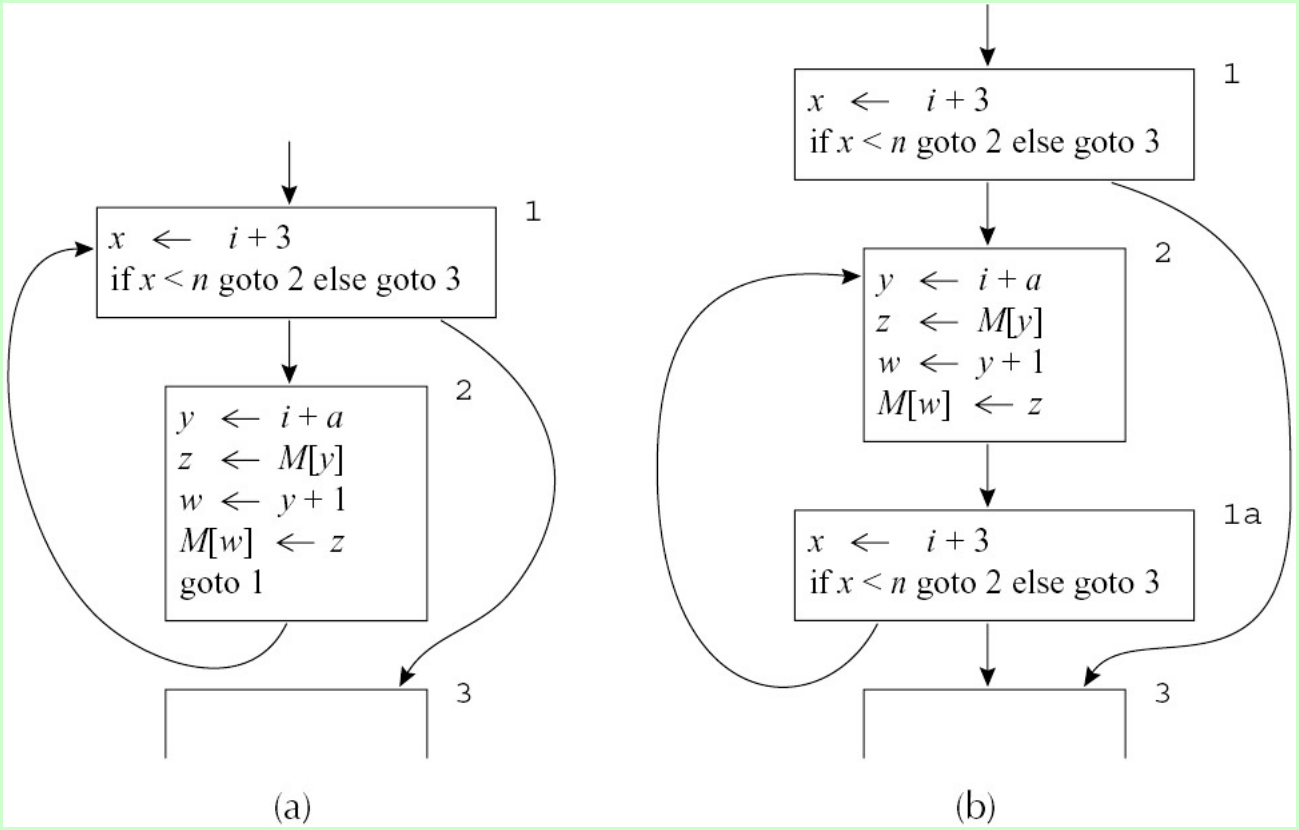
*list_addr + message_offset + (i*message_size+val_offset)*4*

This equals list_addr + message_offset + val_offset*4 + i*message_size*4

Exposing loop invariant operations

PURDUE UNIVERSITY

Handling the potential zero-trip case



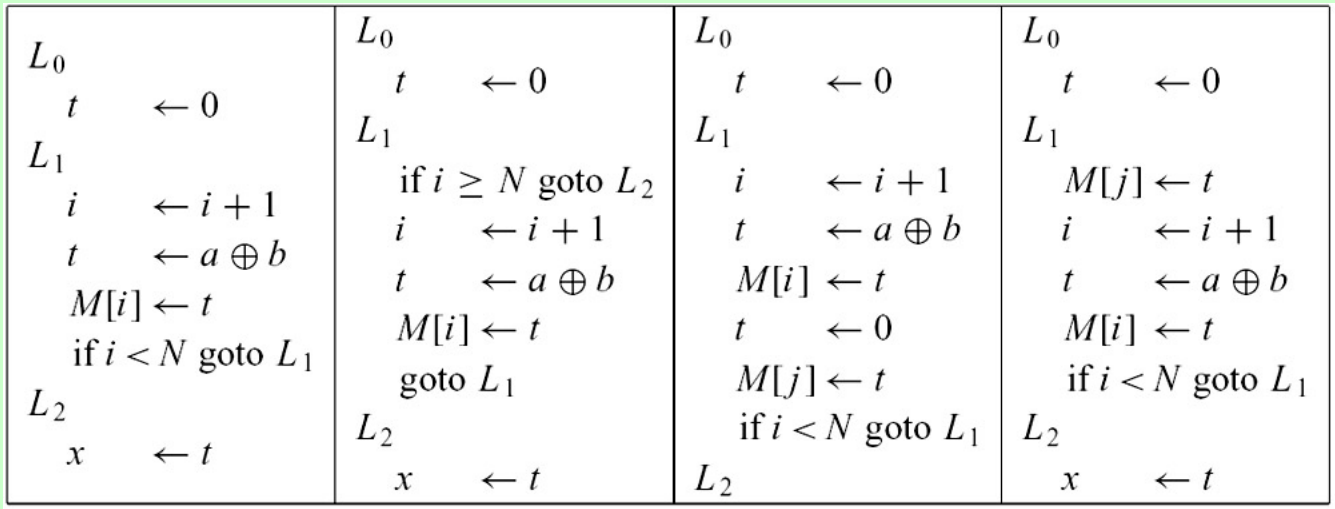
PURDUE UNIVERSITY

Loop invariant code motion

- We cannot always tell if a will have the same value every time, so as usual we will conservatively approximate. The definition d: t ← $a_1 \oplus a_2$ is loop invariant within loop L if, for each operand a_i ,

1. a_i is a constant,
2. *or* all the definitions of a_i that reach *d* are outside the loop,
3. *or* only one definition of a_i reaches *d*, and that definition is loop-invariant.

The transformed code must preserve the def-use relationship. See the next cases:



(a) Hoist (b) Don't (c) Don't (d) Don't

PURDUE UNIVERSITY

Sparse Conditional Constant Propagation

We study the issue of constant propagation and the use of SSA for constant propagation, using an algorithm due to Wegman & Zadeck [ACM TOPLAS 1991].)

What is *constant propagation?*

For each use of a variable, find out whether its value remains constant no matter how the execution path leads to this use. We also want to know that constant value.

```
if i > m error();
```

Is m a constant? If so, what is it's value?

Why Do We Care?

- Performance:
  - If the constant is small enough, it can be embedded within the instruction ($immediate\ operand$). No load from memory is needed.
  - Most $scheduling$ problems can be made easier if key parameters are know constant values.
- Simplifying the code:
  - Constant propagation can expose $dead\ code$, code segments which will never be executed. This reduces the code size.

Scope

- Here we consider scalar variables only.
- Assuming no aliases.
- At the end of constant propagation, each variable, wherever it appears in the program, should be marked either as $constant$ or as nonconstant . In the terminology of lattice theory , the nonconstant value is called $bottom$.
- Thus, during the propagation, at any time, a variable may be marked as one of the following: `top`, `constant`, or `bottom`.

- The most interesting thing happens when information from different execution paths converge. The following lists a set of such $meet\ rules$.

---

a	$\cap$	top	$\longrightarrow$ a
a	$\cap$	bottom	$\longrightarrow$ bottom
constant	$\cap$	constant	$\longrightarrow$ constant (if the values are equal)
constant	$\cap$	constant	$\longrightarrow$ bottom (if the values are unequal)

Here top indicates that nothing is known about the variable yet.

Expression Rules

Suppose a number of variables appear in a certain basic block. Each variable has a certain marking at the entry of the basic block.

- If a variable is not modified in the basic block, then its marking is unchanged.
- If a variable is updated with the value of an expression, then
  - if any variable in that expression has the $bottom$ marking, then the modified variable will also have the $bottom$ marking; (Are we potentially missing any constants by doing this?)

– if all variables used in the expression are constants, then the modified variable will also have the $constant$ marking, the constant value can be determined by evaluating the expression (i.e. by constant folding);
– otherwise, the reassigned variable will have the top marking.

Kildall's Algorithm

Simple constants can be recognized by using Kildall's iterative propagation framework in a simplistic way:

- The condition expression of any conditional branch is not examined. Hence, all branch targets are considered possibly taken. This is despite that possibility that the branch condition may be constant and hence only one branch target is possible.
- Only one value for each variable is maintained along each path in the program. (If a variable can have two possible values, thus not a constant, then besides being marked as *bottom*, the two possible values are not recorded.)

All variables in all basic blocks are initially marked as top, except for the start node in the CFG, in which all variables are marked as $bottom$. We optimistically assume all variables to be constant in all basic blocks except the start node.

Since each variable can only have its lattice value lowered twice, each node may be visited at most $2V$ times, where V is the number of variables.

The time required for Kildall's algorithm is $O(E \times V)$ node visits, and V operations during each node visit. This results in a running time of $O(E \times V^2)$ in the worst case. The space required is $O(N \times V)$.

Wegbreit [IEEE SE 1975] found that, by symbolically executing expressions (including branch conditionals) to recognize $unexecutable$ flow edges, additional constants, called $conditional\ constants$ (CC), can be discovered. The worst-case complexity for doing that is also $O(E \times V^2)$.

W & Z's CC algorithm [TOPLAS 91], here called $sparse\ CC$ algorithm, adopts Reif and Lewis Idea to use def-use chains to propagate constants, instead of propagating V variables through the flow graph.

Further, they use SSA to simplify the def-use chains and examine branch conditions to catch many cases of $conditional\ constants$.

A trivial example:

Example 1:

```
i ← 1
j ← 2
if j = 2
  then i ← 3
endif
... ← i
```

More realistic examples of conditional constants:

Example 2:

```
i ← 1
j ← 1
if i = j
  then i ← i+1
endif
... ← i
```

Example 3:

```
read(i)
j ← 2
if i = j
  then
  ... ← i
endif
```

In order to catch those more realistic conditional constants shown in the last two examples, extra nodes are inserted betwen a conditional which performs an $equality\ test$ (e.g. $i = j$) and the conditional branch targets.

In each of these extra nodes, artificial assignments are inserted to introduce def-use edges which can then be examined during propagation. (e.g. $i \longleftarrow j$)

Some terminologies

- An edge in the def-use chains is called a $root\ edge$ if the source of the edge is not the sink of any other edge (e.g. x := 100).
- Each def or use side is marked by its $level$, which says whether it is `top`, `bottom`, or a constant. Each site known to be constant is also marked by its $constant\ value$. Similarly each join node is marked by its $level$ and $value$.

- In order to differential symbolic values of variables, we can first transform the program into a static single assignment (SSA) form
- Recall that we motivated the SSA transformation when we try to recognize congruent expressions
- SSA will be discussed in the next lecture
  - Computation of SSA is based dominance frontiers

PURDUE
UNIVERSITY