19. Data Cache Optimizations

- **Locality Optimizations**

  Cluster accesses of data values both spacially (within a cache line) and temporally (for repeated use). *Loop interchange* **and** *loop tiling* **improve temporal locality.**

- **Conflict Optimizations**

  Adjust data locations so that data used consecutively and repeatedly don't share the same cache location.

20. Instruction Cache Optimizations

Instructions that are repeatedly executed should be accessed from the instruction cache rather than the secondary cache or memory. Loops and "hot" instruction sequences should fit within the cache.

Temporally close instruction sequences should not map to conflicting cache

Basic Blocks

A basic block is a linear sequence of instructions containing no branches except at the very end.

A basic block is always executed sequentially as a unit.

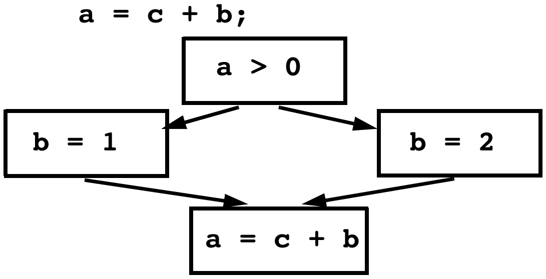
Control Flow Graphs

A Control Flow Graph (CFG) models possible execution paths through a program.

Nodes are basic blocks and arcs are potential transfers of control.

For example,

```
if (a > 0)
        b = 1;
 else   b = 2;
 a = c + b;
```



For a Basic Block b:

Let Preds(b) = the set of basic blocks that are Immediate Predecessors of b in the CFG.

Let Succ(b) = the set of basic blocks that are Immediate Successors to b in the CFG.

Data Flow Problems

A data flow problem is a program analysis computed on a control flow graph.

A data flow problem may be *forward* (following a program's control flow) or *reverse* (opposite a program's control flow).

Informally, forward analyses "remember the past" while reverse analyses "predict the future."

Some analyses determine that an event *may* have occurred, while others determine that an event *must* have occurred.

Some analyses compute a set of values, while others are Boolean-valued.

Two important data flow problems are *Reaching Definitions* and *Liveness*.

For a given use of a variable v reaching definitions tell us which assignments to v may reach (affect) the current value of v. Reaching definition analysis is useful in both optimization and debugging.

Liveness analysis tells us at a particular point in a program whether the current value of variable v will ever be used. A variable that is not live is dead. A dead value need not be kept in memory, or perhaps even be computed.

Reaching Definitions

For a Basic Block b and Variable V:

Let DefsIn(b) = the set of basic blocks that contain definitions of V that reach (may be used in) the beginning of Basic Block b.

Let DefsOut(b) = the set of basic blocks that contain definitions of V that reach (may be used in) the end of Basic Block b.

The sets Preds and Succ are derived from the structure of the CFG.

They are given as part of the definition of the CFG.

DefsIn and DefsOut must be computed, using the following rules:

1. If Basic Block b contains a definition of V then
   DefsOut(b) = {b}
2. If there is no definition to V in b then
   DefsOut(b) = DefsIn(b)
3. For the First Basic Block, b_0:
   $DefsIn(b_0) = \phi$
4. For all Other Basic Blocks
   $$DefsIn(b) = \bigcup_{p \in Preds(b)} DefsOut(p)$$

Liveness Analysis

For a Basic Block b and Variable V:

LiveIn(b) = true if V is Live (will be used before it is redefined) at the beginning of b.

LiveOut(b) = true if V is Live (will be used before it is redefined) at the end of b.

LiveIn and LiveOut are computed, using the following rules:

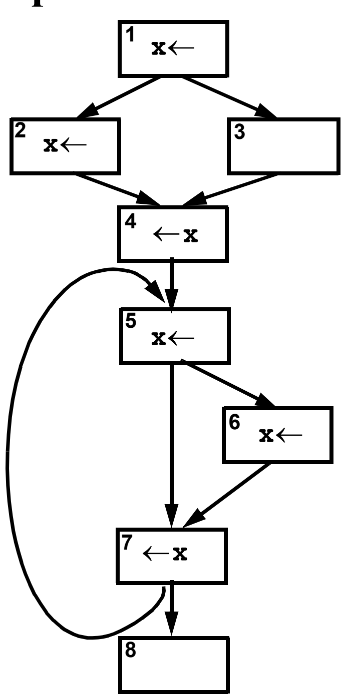
1. If Basic Block b has no successors then
LiveOut(b) = false

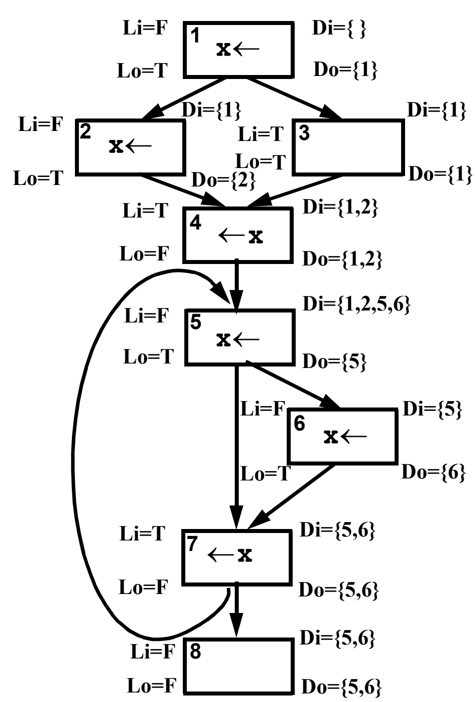
2. For all Other Basic Blocks

$$\text{LiveOut(b)} = \bigvee_{s \in \text{Succ(b)}} \text{LiveIn(s)}$$

3. LiveIn(b) =
If V is used before it is defined in Basic Block b
Then true
Elsif V is defined before it is used in Basic Block b
Then false
Else LiveOut(b)

Example





Reading Assignment

- Section 14.3 - 14.4 of CaC

Data Flow Frameworks

- **Data Flow Graph:**

  Nodes of the graph are basic blocks or individual instructions.

  Arcs represent flow of control.

  Forward Analysis:

  Information flow is the same direction as control flow.

  Backward Analysis:

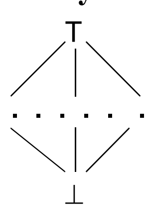
  Information flow is the opposite direction as control flow.

  Bi-directional Analysis:

  Information flow is in both directions. (Not too common.)

- **Meet Lattice**

  Represents solution space for the data flow analysis.



- **Meet operation**
  (And, Or, Union, Intersection, etc.)

  Combines solutions from predecessors or successors in the control flow graph.

- **Transfer Function**

  Maps a solution at the top of a node to a solution at the end of the node (forward flow)
  or
  Maps a solution at the end of a node to a solution at the top of the node (backward flow).

Example: Available Expressions

This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.

The Meet Lattice is:

T (Expression is Available)
|
F (Expression is Not Available)

As initial values, at the top of the start node, nothing is available. Hence, for a given expression,

AvailIn(b_0) = F

We choose an expression, and consider all the variables that contribute to its evaluation.

Thus for e_1=a+b-c, a, b and c are e_1's *operands*.

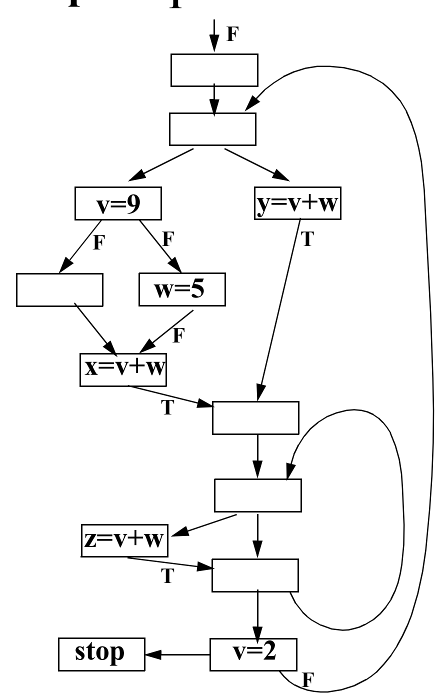
The transfer function for e_1 in block b is defined as:

If e_1 is computed in b after any
assignments to e_1's operands in b
Then AvailOut(b) = T
Elsif any of e_1's operands are changed
after the last computation of e_1 or
e_1's operands are changed without
any computation of e_1
Then AvailOut(b) = F
Else AvailOut(b) = AvailIn(b)

The meet operation (to combine solutions) is:

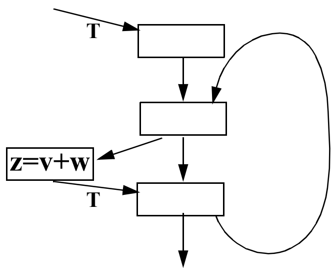
$$\text{AvailIn}(b) = \underset{p \in \text{Pred}(b)}{\text{AND}} \text{AvailOut}(p)$$

Example: e_1=v+w



Circularities Require Care

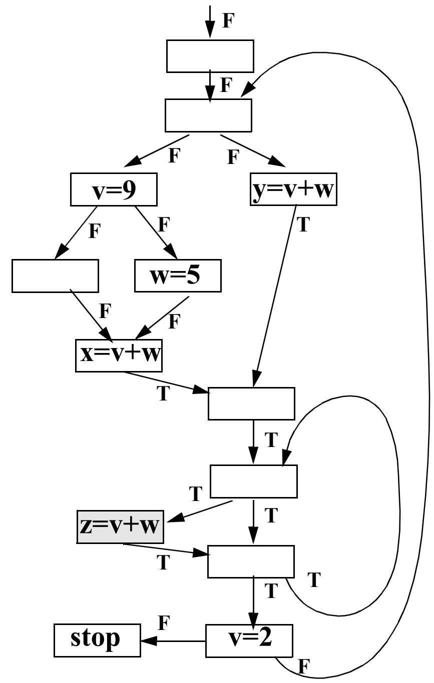
Since data flow values can depend on themselves (because of loops), care is required in assigning initial "guesses" to unknown values.



Consider

If the flow value on the loop backedge is initially set to false, it can never become true. (Why?)

Instead we should use True, the *identity* for the AND operation.



Very Busy Expressions

This is an interesting variant of available expression analysis.

An expression is *very busy* at a point if it is *guaranteed* that the expression will be computed at some time in the future.

Thus starting at the point in question, the expression must be reached before its value changes.

Very busy expression analysis is a backward flow analysis, since it propagates information about future evaluations backward to "earlier" points in the computation.

The meet lattice is:

T (Expression is Very Busy)
|
F (Expression is Not Very Busy)

As initial values, at the end of all exit nodes, nothing is very busy. Hence, for a given expression,

$\text{VeryBusyOut}(b_{last}) = F$

The transfer function for e_1 in block b is defined as:

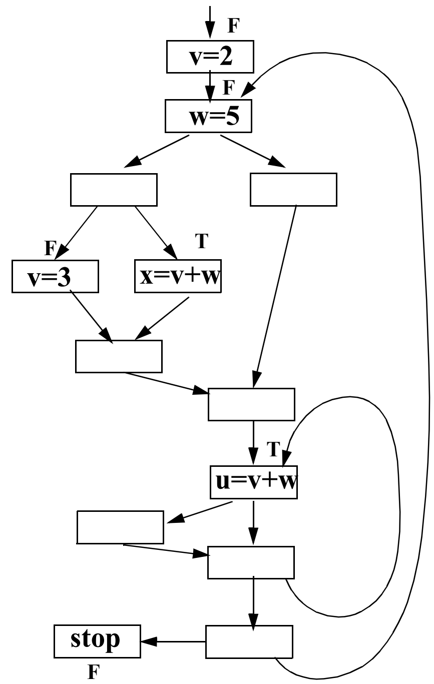
If e_1 is computed in b before any of its operands
Then VeryBusyIn(b) = T
Elsif any of e_1's operands are changed before e_1 is computed
Then VeryBusyIn(b) = F
Else VeryBusyIn(b) = VeryBusyOut(b)

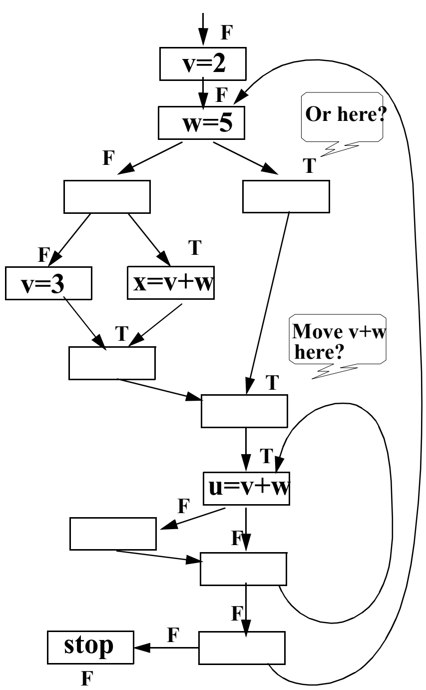
The meet operation (to combine solutions) is:

VeryBusyOut(b) =

$$\underset{s \in \text{Succ}(b)}{\text{AND}} \text{VeryBusyIn}(s)$$

Example: e_1=v+w





Identifying Identical Expressions

We can hash expressions, based on hash values assigned to operands and operators. This makes recognizing potentially redundant expressions straightforward.

For example, if H(a) = 10, H(b) = 21 and H(+) = 5, then (using a simple product hash),
H(a+b) = $10 \times 21 \times 5$ Mod TableSize

Effects of Aliasing and Calls

When looking for assignments to operands, we must consider the effects of pointers, formal parameters and calls.

An assignment through a pointer (e.g, `*p = val`) *kills* all expressions dependent on variables `p` might point too. Similarly, an assignment to a formal parameter kills all expressions dependent on variables the formal might be bound to.

A call kills all expressions dependent on a variable changeable during the call.

Lacking careful alias analysis, pointers, formal parameters and calls can kill all (or most) expressions.

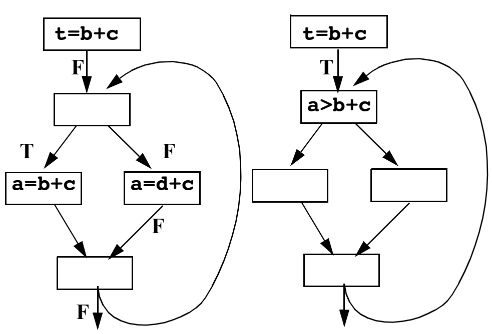
Very Busy Expressions and Loop Invariants

Very busy expressions are ideal candidates for invariant loop motion.

If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.


```
for (...) {
 if (...)
      a=b+c;
 else a=d+c;}
```

```
for (...) {
 if (a>b+c)
      x=1;
 else x=0;}
```



`b+c` is not very busy at loop entrance

`b+c` is very busy at loop entrance

Reaching Definitions

We have seen reaching definition analysis formulated as a set-valued problem. It can also be formulated on a per-definition basis.

That is, we ask “What blocks does a particular definition to v reach?”

This is a boolean-valued, forward flow data flow problem.

Initially, DefIn(b_0) = false.

For basic block b:

DefOut(b) =
If the definition being analyzed is the last definition to v in b
Then True
Elsif any other definition to v occurs in b
Then False
Else DefIn(b)

The meet operation (to combine solutions) is:

$$\text{DefIn}(b) = \underset{p \in \text{Pred}(b)}{\text{OR}} \text{DefOut}(p)$$

To get all reaching definition, we do a series of single definition analyses.

Live Variable Analysis

This is a boolean-valued, backward flow data flow problem.

Initially, LiveOut(b_{last}) = false.

For basic block b:

LiveIn(b) =
If the variable is used before it is defined in b
Then True
Elsif it is defined before it is used in b
Then False
Else LiveOut(b)

The meet operation (to combine solutions) is:

$$\text{LiveOut}(b) = \underset{s \in \text{Succ}(b)}{\text{OR}} \text{LiveIn}(s)$$

Bit Vectoring Data Flow Problems

The four data flow problems we have just reviewed all fit within a *single* framework.

Their solution values are Booleans (bits).

The meet operation is And or OR.

The transfer function is of the general form

Out(b) = (In(b) - $Kill_b$) U Gen_b

or

In(b) = (Out(b) - $Kill_b$) U Gen_b

where $Kill_b$ is true if a value is "killed" within b and Gen_b is true if a value is "generated" within b.

In Boolean terms:

Out(b) = (In(b) AND Not $Kill_b$) OR Gen_b

or

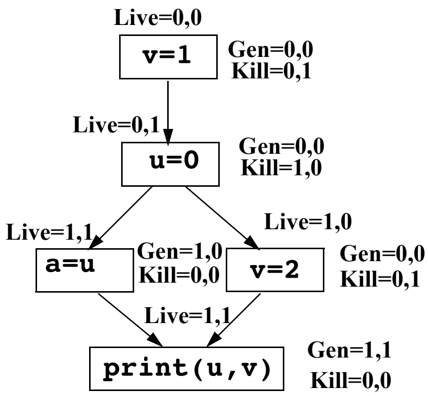
In(b) = (Out(b) AND Not $Kill_b$) OR Gen_b

An advantage of a bit vectoring data flow problem is that we can do a series of data flow problems "in parallel" using a bit vector.

Hence using ordinary word-level ANDs, ORs, and NOTs, we can solve 32 (or 64) problems simultaneously.

Example

Do live variable analysis for u and v, using a 2 bit vector:



We expect *no variable* to be live at the start of b_0. (Why?)