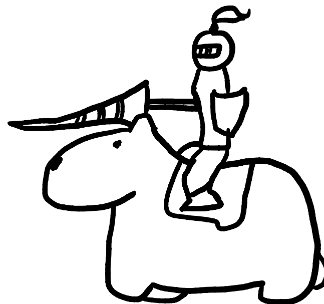
CMU 15-745 S21

NOTES

wanshenl

Prof: Todd Mowry

Book: Compilers 2E
Aho, Lam, Sethi, Ullman



CMU 15-745 Notes, Prof: Todd Mowry, Book: Dragon book 2E

Chapter 9.1 Principal Sources of Optimizations

- **Code improvement** = { elimination of unnessary instructions in object code; replacing instruction sequence s_1 with faster equivalent s_2 }
  - Local = within basic block
  - Global = across basic blocks, mostly based on **data-flow analysis** (later)

note: global is misleading, still only within procedure, across procedures is called interprocedural analysis

- **Three-address statements**
  - At most three operands A=B op C
  - Easier to translate to assembly
  - Easier to detect common subexpressions (later)

  e.g. x=a[i] → t6=4*i, x=a[t6] for i a 4-byte integer

  highlights the fact that a high-level language will result in unavoidable redundancy in computing 4*i for every array offset

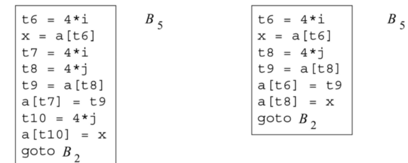


Figure 9.4: Local common-subexpression elimination

- **Common subexpressions**

  An occurrence of an expression E is a common subexpression if:
  ① E was previously computed
  ② values of variables in E have not changed since previous computation
  - However, while stuff like 4*i can be eliminated, array accesses like a[t] may not be if it goes through a basic block which assigns to a.

- **Copy propagation**

  Assignments of the form u=v are called **copy statements**.
  - Underlying idea: use v for u wherever possible after u=v

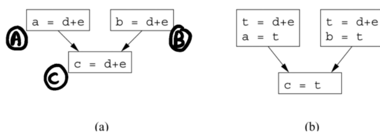


Figure 9.6: Copies introduced during common subexpression elimination

Note that c=d+e cannot be replaced by c=a or c=b since control may reach © after Ⓐ or Ⓑ.

- **Dead-Code Elimination**
  - Variable v is **live** at a point in a program if its value can be used subsequently and is **dead** at that point otherwise.
  - **Constant folding** is deducing at compile time that a variable is a constant and using that instead.

  e.g. if (DEBUG) →(constant folding) if (false) →(dead code elimination) nothing

- **Code Motion**
  - Programs spend a bulk of their time in loops

  ⇒ might improve running time by decreasing code in loop, increasing code outside loop

  e.g. while (i<=limit-2) → t=limit-2; while (i<=t)

Chapter 9.1 Continued

- Induction variables
  Variable x is an **induction variable** if ∃ positive or negative constant c such that each time x is assigned, its value increases by c.
  - **Strength reduction** is replacing an expensive operation by a cheaper one.

  e.g.
  ```
  j=j-1            t4=4*j
  t4=4*j    →        :
                   j=j-1
                   t4=t4-4
  ```

Lecture 2/2

- **Basic block** = sequence of 3-address statements
  - Only the first statement can be reached from outside the block, no branches into the middle
  - All statements are executed consecutively if the first one is, no branches or halts (except maybe at end)
  - Maximal – cannot be made larger without violating conditions

- **Flow graphs**
  - Nodes: basic blocks
  - Edges: $B_i \rightarrow B_j$ iff B_j can follow B_i immediately in some execution
  - Block led by first statement of program is start (or entry) node.

- **Code hoisting**
  Eliminate copies of identical code on parallel paths in a flow graph to reduce code size

- **Elimination of loop index**
  - Replace termination by tests on other induction variables

---

Chapter 8.4 Basic Blocks and Flow Graphs

- Basic blocks
  - Subtlety: interrupts can be ignored.
    If an interrupt occurs and is
    - handled → control will come back as if control never deviated
    - not handled → program crashes with an error anyway

- **Partitioning** three-address statements into basic blocks

INPUT	A sequence of three-address statements
OUTPUT	A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.
METHOD	① Identify **leaders**, the first instructions in some basic block. · The first three-address instruction in the intermediate code is a leader. · Any instruction that is the target of a conditional or unconditional jump is a leader. · Any instruction that immediately follows a conditional or unconditional jump is a leader. ② For each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

Chapter 8.4 cont.

· Next-Use Information

does used ⇔ live? not exactly. liveness propagates, use is local.

If { stmt i assigns x, stmt j has x as an operand, ∃ control $i \xrightarrow{\text{no assign}(x)} j$ } Then { statement j **uses** the value of x computed at statement i; x is **live** at statement i }

· To determine liveness and next-use information.

INPUT	Basic block B, assuming symbol table initially shows all non-temporary variables in B as being live on exit
OUTPUT	At each $i: x=y+z$ in B, attach to i the liveness and next-use information of $x, y,$ and z.
METHOD	① Start at the last statement in B. ② Scan backwards to the beginning of B. ③ At each $i: x=y+z$ in B, [1] Attach next use and liveness of x,y,z from symbol table to i [2] In symbol table, set x to not live and no next use [3] In symbol table, set y and z to live and next uses of y and z to i. Order matters! Consider $x=x+x$.

· Flow graphs already covered

- Just note that **logical** representations are better as we may frequently change the number and type of instructions in a basic block

· Loops

A set of nodes L in a flow graph is a **loop** if L contains **loop entry** node e where:

- e is not ENTRY the entry point to the flow graph.
- No node in L besides e has a predecessor outside L.
- Every node in L has a nonempty path completely within L to e.

Chapter 8.5 Optimization of Basic Blocks

· Constructing DAGs (directed acyclic graphs) for basic blocks

① A node for each initial value of variables in basic block
② A node N for each statement s.
N's children are nodes that correspond to statements that last defined (prior to s) operands used by s.
③ N is labeled by the operator at s.
④ The list of variables for which it is the last definition in the block is attached to N.

· With the DAG, we can:

- eliminate local common subexpressions
- eliminate dead code
- reorder independent statements
- apply algebraic laws for simplification

Chapter 8.5 cont.

- **Local common subexpressions** — **value-number method**
  - Before adding new node M, check if exists node N with same children, in same order, with same operator.

- **Dead code elimination**
  - Repeat: delete any root with no live variables

- **Algebraic identities**
  - Some examples

    $x+0=0+x=x$, $x/1=x$ } **algebraic**

    $x^2=x*x$, $2*x=x+x$ } local **reduction in strength**, replacing expensive op with cheaper

- Representation of array references

  $x=a[i]$
  $a[j]=y$
  $z=a[i]$
  } what if $j=i$? $a[i]$ <u>cannot</u> be naively optimized

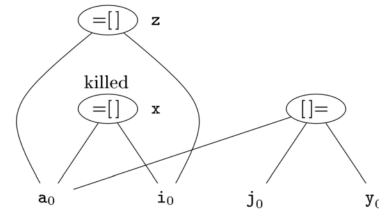
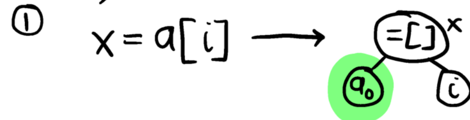


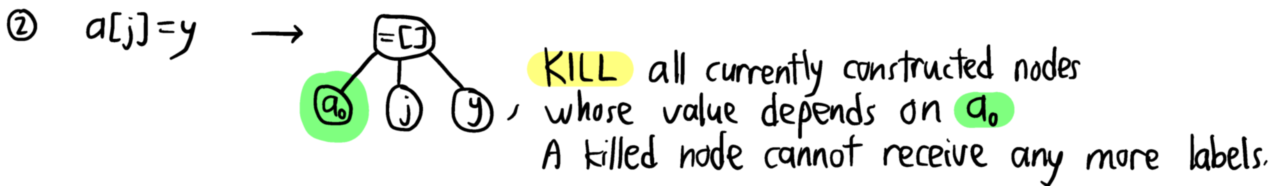
Figure 8.14: The DAG for a sequence of array assignments

Instead,

① $x=a[i]$ →



② $a[j]=y$ →

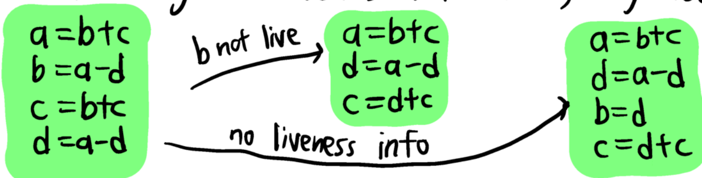


KILL all currently constructed nodes whose value depends on a_0
A killed node cannot receive any more labels.

- Pointer assignments and procedure calls
  - Assigning indirectly through a pointer

    $x=*p$ uses every variable
    $*q=y$ assigns every variable
    } (=*) kills all the other nodes constructed so far in the DAG
  - Similarly procedures called in the scope of x both uses and kills x's node

- Reassembling basic blocks from DAGs
  - Prefer to compute results into variables that are live on exit from the block
  - But if no global liveness information, may need copy statements

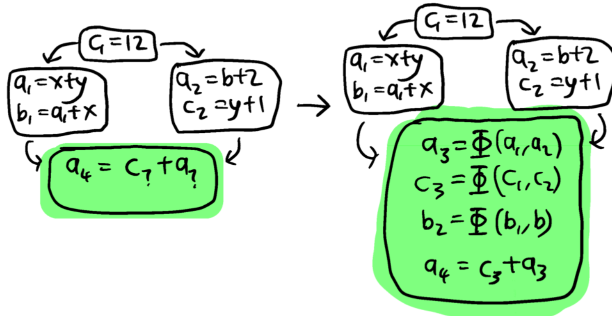


- Rules for DAG reconstruction
  - ⓪ Respect DAG node order
  - ② Assignments to array must follow previous assignments
  - ③ Evaluations of array elements must follow all previous assignments, two evaluations on same array can be reordered if both don't cross assignments
  - ④ Variable use must follow all previous procedure calls or indirect assignments
  - ⑤ Any procedure call or indirect assignments must follow all previous evaluations

Lecture 2/3

- Φ functions in SSA and LLVM (for Assignment 1)
  - Motivation: where is a variable defined or used?
  - Traversing directly between related uses and definitions would enable sparse code analysis
  - Appearances of the same variable name may be unrelated

- Single Static Assignment (SSA)
  - Every variable is assigned a value at most once
  - But what about join nodes?

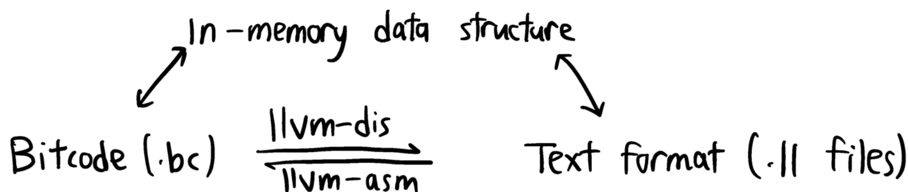


  - Treat Φ like any other arithmetic function for now

Lecture 2/4

- LLVM compiler system
  - Infrastructure: reusable components for building compilers
  - Framework: E2E compilers built with above
  - Three phase design
  - Optimizer is series of analysis (doesn't change code) and optimization (changes code) passes

- LLVM IR



The bitcode and text formats are lossless!

- Program Structure
  - Module F⇄F⇄F...
  - Function BB⇄BB⇄BB...
  - Basic block I⇄I⇄I...

- LLVM Pass Manager
  - Compiler is organized as a series of passes
  - Four types of passes

Chapter 9.2 Introduction to Data-Flow Analysis

- **Data-flow analysis**
  A body of techniques that derive information about the flow of data along program execution paths.

- **Execution Path**
  The path from point p_1 to point p_n is the sequence of points $p_1, p_2, \ldots, p_n$ where for each $i = 1, 2, \ldots, n-1$:
  - p_i precedes a statement and p_{i+1} immediately follows the same statement
  - p_i ends some block and p_{i+1} begins a successor block

- **Imperfect representations**
  - Not possible to track all program states
  - Definitions which may reach a program point along some path are **reaching definitions**.

- **Data-flow analysis schema**
  - Every program point has a **data-flow value** associated which represents {all possible program states}
  - **Domain**: set of possible data-flow values
  - **IN[s]** : data-flow values before statement s
  - **OUT[s]**: data-flow values after statement s

- **Data-flow problem**
  - Find a solution to the set of constraints on IN[s] and OUT[s] for all statements s
  - Two sets of constraints
    - Based on semantics (Transfer functions)
    - Based on control flow (Control-flow constraints)

- **Transfer Functions**
  - e.g., if $a = v$ $\xrightarrow{b=a}$ then $a = v$, $b = v$
  - Information can propagate forward or backward along the execution path
  - **Forward-flow** problem
    - $OUT[s] = f_s(IN[s])$
  - **Backward-flow** problem
    - $IN[s] = f_s(OUT[s])$

  f_s = transfer function of statement s

- **Control-flow constraints**
  - Within basic block B containing $[s_1, s_2, \ldots, s_n]$,
    - $IN[s_{i+1}] = OUT[s_i]$ for all $i = 1, 2, \ldots, n-1$
  - Between basic blocks, restate schema in terms of data-flow values entering/leaving block
    - $IN[B] = IN[s_1]$
    - $OUT[B] = OUT[s_n]$
    - $f_B = f_{s_n} \circ f_{s_{n-1}} \circ f_{s_{n-2}} \circ \cdots \circ f_{s_2} \circ f_{s_1}$
    - $OUT[B] = f_B(IN[B])$

- Dataflow equations usually don't have a unique solution
  - Instead, find the most "precise" satisfying control-flow and transfer constraints

Chapter 9.2 cont.

Reaching Definitions

- Definition d **reaches** point p if there is a path from the point immediately following d to p such that d is not killed along the path.
- A definition of variable x is **killed** if there is any other definition of x along the path.
- Application: possible use before define
  - Add a dummy definition for each variable on flow graph entry
  - If dummy might be used, then there *might* be use before define (only might!)
- Note that all possible inaccuracies are safe/conservative
- Transfer function

  example. definition $d: u = v + w$

  $f_d = gen_d \cup (x - kill_d)$ transfer function of definition d

  $gen_d = \{d\}$ the set of definitions generated by the statement

  $kill_d$ the set of all other definitions of u in the program

  note that kill happens before gen, so a variable can be both killed and gend with gen taking priority!

  This also applies to basic blocks.

  downwards exposed: gen set definitions that are visible immediately after the block, a definition is downwards exposed in a basic block only if it is not killed by a subsequent definition to the same variable inside the basic block
- Control-flow equations
  - Note $OUT[P] \subseteq IN[B]$ whenever $P \rightsquigarrow B$
  - Union is the **meet operator** for reaching definitions

    $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$

- **Reaching definitions problem**
  - $OUT[entry] = \emptyset$
  - $\forall B \neq entry$,

    $OUT[B] = gen_B \cup (IN[B] - kill_B)$

    $IN[B] = \bigcup_{P \text{ predecessor of } B} OUT[P]$

- **Iterative algorithm**

INPUT	Flow graph where $kill_B$ and gen_B computed for each B
OUTPUT	$IN[B]$ and $OUT[B]$
METHOD	① $\forall B$, $OUT[B] = \emptyset$ ② while (OUT changes): $\forall B \neq entry$, $IN[B] = \bigcup_{P \text{ predecessor}(B)} OUT[P]$ $OUT[B] = gen_B \cup (IN[B] - kill_B)$ } empirically <5 iterations on avg!

$IN[EXIT] = \emptyset$;
for (each basic block B other than EXIT) $IN[B] = \emptyset$;
while (changes to any IN occur)
  for (each basic block B other than EXIT) {
    $OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$;
    $IN[B] = use_B \cup (OUT[B] - def_B)$;
  }

Figure 9.16: Iterative algorithm to compute live variables

Live Variable Analysis

- Could the value of variable x at point p be used in some path $p \rightsquigarrow$.?

  If yes, x is **live** at p.

  If no, x is **dead** at p.
- Application: register allocation. Don't store dead values, prefer overwriting dead values.
- Note that this is a backward problem, so initialize IN[exit] instead of OUT[entry], interchange IN and OUT in iterative algorithm.

Chapter 9.2 Cont.

- **Available Expressions**
  - An expression $x \oplus y$ is **available** at a point p if
    - every path entry $\rightsquigarrow p$ evaluates $x \oplus y$
    - after the last $x \oplus y$ evaluation prior to p, there are no subsequent assignments to x or y
  - A block **kills** expression $x \oplus y$ if it may assign x or y and does not recompute $x \oplus y$
  - A block **generates** expression $x \oplus y$ if it definitely evaluates $x \oplus y$ and does not subsequently define x or y.

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e_gen_B \cup (x - e_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

Lecture 2/9

- **Locally exposed use** in a basic block is the use of a data item which is not preceded in the basic block by a definition of the data item
- **Locally available definition** = last definition of data item in basic block

Reaching Definitions: Worklist Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
    out[Entry] = Ø           // can set out[Entry] to special def
                             // if reaching then undefined use
    For all nodes i
        out[i] = Ø           // can optimize by out[i]=gen[i]
    ChangedNodes = N

// iterate
    While ChangedNodes ≠ Ø {
        Remove i from ChangedNodes
        in[i] = U (out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = f_i(in[i])      // out[i]=gen[i]U(in[i]-kill[i])
        if (oldout ≠ out[i]) {
            for all successors s of i
                add s to ChangedNodes
        }
    }
```

Chapter 9.3 Foundations of Data-Flow Analysis

- **Data-flow analysis framework**

  $(D, V, \wedge, F)$

  D: direction of data-flow, {forward/backward}
  V: domain of values
  $\wedge$: meet operator
  F: family of transfer functions $V \to V$

- **Semilattice**

  $(V, \wedge)$

  where $\forall x,y,z \in V$
  - idempotent $x \wedge x = x$
  - commutative $x \wedge y = y \wedge x$
  - associative $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - has top element $\top$ where $\forall x \in V$ $\top \wedge x = x$
  - optionally has bottom element $\bot$ where $\forall x \in V$ $\bot \wedge x = \bot$

- **Partial order**

  $\leq$ is a partial order on V if $\forall x,y,z \in V$,
  - reflexive $x \leq x$
  - antisymmetric $x \leq y$ and $y \leq x \Rightarrow x = y$
  - transitive $x \leq y$ and $y \leq z \Rightarrow x \leq z$

- **Partially ordered set** (poset)

  $(V, \leq)$ is a poset
  $<$ may be defined, where $x < y$ iff $x \leq y$ and $x \neq y$

- **Partially ordered semilattice**

  Given $(V, \wedge)$, $\forall x,y \in V$ $x \leq y$ iff $x \wedge y = x$
  $\wedge$ idempotent commutative associative $\Rightarrow$ $\leq$ reflexive antisymmetric transitive

- **Greatest lower bound** (glb)
  - Given $(V, \wedge)$, glb of $x, y \in V$ is g satisfying
    - $g \leq x$
    - $g \leq y$
    - If z any element satisfying $z \leq x$ and $z \leq y$ then $z \leq g$
  - **$x \wedge y$ is the only glb** of x and y

- **Least upper bound** (lub) analogous, forming **join semilattices** instead of **meet semilattices**
- A **lattice** has both meet $\wedge$ and join $\vee$

Chapter 9.3 cont.

- **Product lattices**
  - Given $(A, \wedge_A)$ and $(B, \wedge_B)$ (semi)lattices, their product lattice is

    $$(A \times B, \wedge_p)$$

    where $(a,b) \wedge_p (a',b') = (a \wedge_A a', b \wedge_B b')$
    and $(a,b) \le (a',b')$ iff $a \le_A a'$ and $b \le_B b'$ ⟲ follows from above

- **Height of semilattice**
  - Helps us understand rate of data-flow analysis convergence
  - An **ascending chain** in poset $(V, \le)$ is a sequence $x_1 < x_2 < \cdots < x_n$
  - The **height** of a semilattice is the largest number of $<$ relations in any ascending chain
    Todd prefers $>$ and descending chain

- **Transfer Functions**
  - Family $F: V \to V$ satisfies
    - Identity I where $\forall x \in V \;\; I(x) = x$
    - Closed under composition, for any $f, g \in F \;\; h(x) = g(f(x)) \in F$

- **Framework Properties** $(D, F, V, \wedge)$
  - **Monotone** if $\forall x, y \in V$ and $f \in F$, $x \le y \Rightarrow f(x) \le f(y)$ (equivalently $f(x \wedge y) \le f(x) \wedge f(y)$)
  - **Distributive** if $\forall x y \in V$ and $f \in F$, $f(x \wedge y) = f(x) \wedge f(y)$

- **Iterative Algorithm**

Algorithm 9.25: Iterative solution to general data-flow frameworks.

INPUT: A data-flow framework with the following components:

1. A data-flow graph, with specially labeled ENTRY and EXIT nodes,
2. A direction of the data-flow D,
3. A set of values V,
4. A meet operator $\wedge$,
5. A set of functions F, where f_B in F is the transfer function for block B, and
6. A constant value v_{ENTRY} or v_{EXIT} in V, representing the boundary condition for forward and backward frameworks, respectively.

OUTPUT: Values in V for IN$[B]$ and OUT$[B]$ for each block B in the data-flow graph.

METHOD: The algorithms for solving forward and backward data-flow problems are shown in Fig. 9.23(a) and 9.23(b), respectively. As with the familiar iterative data-flow algorithms from Section 9.2, we compute IN and OUT for each block by successive approximation. □

```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (each basic block B other than ENTRY) OUT[B] = ⊤;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ∧_{P a predecessor of B} OUT[P];
6)                  OUT[B] = f_B(IN[B]);
            }
```

(a) Iterative algorithm for a forward data-flow problem.

```
1)  IN[EXIT] = v_EXIT;
2)  for (each basic block B other than EXIT) IN[B] = ⊤;
3)  while (changes to any IN occur)
4)          for (each basic block B other than EXIT) {
5)                  OUT[B] = ∧_{S a successor of B} IN[S];
6)                  IN[B] = f_B(OUT[B]);
            }
```

(b) Iterative algorithm for a backward data-flow problem.

Figure 9.23: Forward and backward versions of the iterative algorithm

Provable Properties:
- Converges $\Rightarrow$ solution to data-flow equations
- Monotone $\Rightarrow$ solution is maximum fixedpoint MFP
- Monotone and finite height $\Rightarrow$ guaranteed to converge

- **Ideal vs MOP vs MFP Solution**, assuming forward

Consider any path $P = entry \to B_1 \to B_2 \to \ldots \to B_{k-1} \to B_k$
with transfer function f_P the composition of $f_{B_1}, f_{B_2}, \ldots, f_{B_{k-1}}$ (note f_{B_k} excluded, path is to beginning of B_k)

$$\text{IDEAL}[B] = \bigwedge_{P:\text{ a possible path from entry to } B} f_P(v_{entry})$$

Since considering all possible execution paths is undecidable, assume every path can be taken (meet over paths)

$$\text{MOP}[B] = \bigwedge_{P:\text{ a path from entry to } B} f_P(v_{entry})$$, note $\text{MOP} \le \text{IDEAL}$

But the flow graph may have cycles, so

MFP:
- basic blocks visited may not be in order of execution
- initialized with safe artificial $\top$ "no info" value

, so $\text{MFP} \le \text{MOP}$

Lecture 2/10

- Speed of convergence depends on visit order

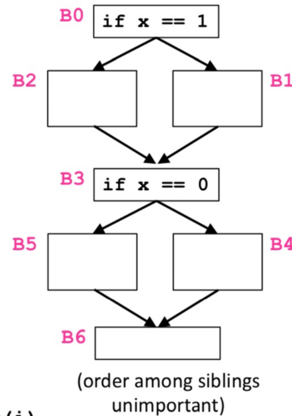
Reverse Postorder

- **Step 1: depth-first post order**

```
main() {
  count = 1;
  Visit(root);
}
Visit(n) {
  for each successor s that
            has not been visited
     Visit(s);
  PostOrder(n) = count;
  count = count+1;
}
```

- **Step 2: reverse order**

```
For each node i
  rPostOrder(i)= NumNodes - PostOrder(i)
```



Carnegie Mellon

15-745: Foundations of Data Flow 29

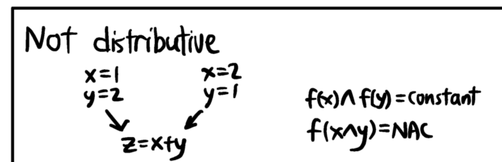
- A good nesting depth in real programs is ≈ 2.75.
  - number of back edges in the path

Lecture 2/11

- LLVM StringRef
- outs(), errs(), null()
- zzz

Chapter 9.4 Constant Propagation

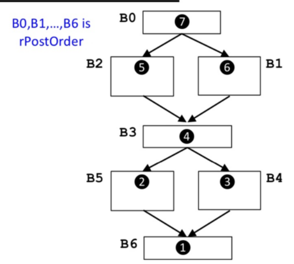
- Properties
  - Unbounded set of possible dataflow values
  - Not distributive
  - Monotone



- Lattice for a single variable contains
  - All type-appropriate constants
  - NAC not a constant
  - UNDEF undefined
    - Safe to optimize when x∧UNDEF, "this random value no worse"

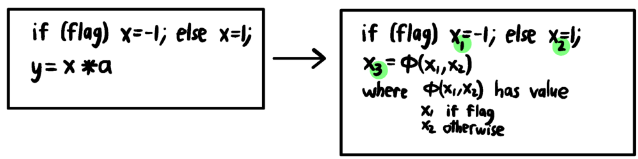
- Mostly common sense

Lecture 2/16



Chapter 6.2.4 Static Single-Assignment Form

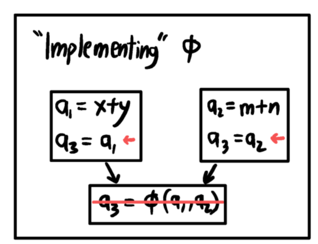
- **Single-Static Assignment** (SSA)
  - IR which facilitates certain code optimizations
  - Two main differences from three-address code
    - All SSA assignments are to variables with **distinct names**
    - **Φ function** to combine control-flow paths



Lecture 2/17

- Recurring theme: knowing where a variable is **defined/used** is very useful
  - Loop invariant code motion
  - Copy propagation
  - Traversing directly between related uses and defs enables **sparse code analysis**
- Appearances of same variable name can be **unrelated**
- Two rarely used solutions
  - **Use-Definition** chains : def of x → all uses of x
  - **Definition-Use** chains : use of x → all reaching def of x
  - N defs, M uses → O(MN) space and time

- **Φ** merges multiple definitions along multiple control paths into a single definition
  - **Syntax trick**, not an actual instruction
  - At a basic block with p predecessors, **$x_{new} \leftarrow \Phi(x_1, x_2, \dots, x_p)$**



- SSA
  1. Each assignment generates a **fresh variable**
  2. At each join point insert Φ functions for **all live variables** with **multiple outstanding defs.**
     - all live variables → **trivial SSA**
     - this addition = **minimal SSA**

Lecture 2/17 cont.

- Goal: doing SSA without doing reaching def/liveness
- Insert Φ for var A in block z iff
  - A was defined more than once before
  - $\exists$ non-empty path $x \rightsquigarrow z$ P_{xz} and non-empty $y \rightsquigarrow z$ P_{yz} where
    - $P_{xz} \cap P_{yz} = \{z\}$
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \rightsquigarrow z$ and $P_{yz} = P_{yr} \to z$
  - Entry block implicitly defines all vars
  - Note $A = \Phi(\cdots)$ is a def of A

- Dominance
  - Block x strictly dominates block w (x sdom w) iff impossible to reach w without passing through x first
  - x dominates w (x dom w) iff x sdom w OR x=w
- Dominance Tree (D-Tree)
  - x sdom w iff x is a proper ancestor of w
- In SSA, definitions dominate uses.
  - x_i used in $x \leftarrow \Phi(x_1, \cdots, x_i, \cdots, x_p)$ $\Rightarrow$ $BB(x_i)$ dominates i^{th} predecessor of $BB(\Phi)$
  - x used in $y \leftarrow \cdots x \cdots$ $\Rightarrow$ $BB(x)$ dominates $BB(y)$
- Dominance Frontier
  - Dominance frontier of node x = {w : x dom pred(w) and !(x sdom w)}
  - The frontier nodes are the ones that need a Φ
  - "Points where paths converge"

Using Dominance Frontier to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DominanceFrontier(defsite)
      - if we haven't put Φ() in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites
- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of Φ() neccesary

Carnegie Mellon
15-745: Intro to SSA 22 Todd C. Mowry

Using Dominance Frontier to Place Φ()

```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}
     defsites[v] ∪= {n}
   }
}
foreach variable v {
   W = defsites[v]
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
     if y ∉ PHI[v] {
       insert "v ← Φ(v,v,…)" at top of y
       PHI[v] = PHI[v] ∪ {y}
       if v ∉ orig[y]: W = W ∪ {y}
     }
   }
}
```

Carnegie Mellon
15-745: Intro to SSA 23 Todd C. Mowry

Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: rename (v)
  - rename(v): replace uses with top of stack
    at def: push onto stack
    call rename(v) on all children in D-tree
    for each def in this block pop from stack

Carnegie Mellon
15-745: Intro to SSA 24 Todd C. Mowry

Computing the Dominance Frontier

The Dominance Frontier of a node x =
{ w | x dom pred(w) AND !(x sdom w)}

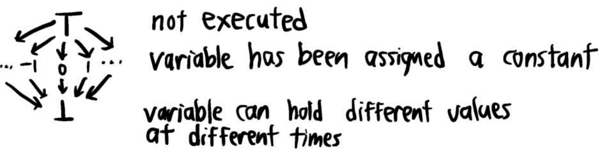
```
compute-DF(n)
        S = {}
        foreach node y in succ[n]
                if idom(y) ≠ n
                        S = S ∪ { y }
        foreach child of n, c, in D-tree
                compute-DF(c)
                foreach w in DF[c]
                        if !n dom w
                                S = S ∪ { w }
        DF[n] = S
```

Carnegie Mellon
Todd C. Mowry 15-745: Intro to SSA 36

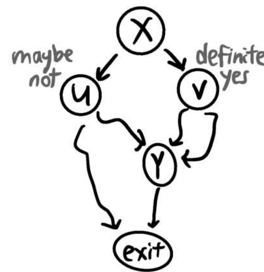
Lecture 2/18

- **Constant Propagation**
  - If $v \leftarrow c$, replace all uses of v with c
  - If $v \leftarrow \phi(c,c,c)$, replace all uses of v with c
- **Copy propagation**
  - Delete $x \leftarrow \phi(y,y,y)$ and replace all x with y
  - Delete $x \leftarrow y$ and replace all x with y
- **Constant folding**, **constant conditions**, etc.

- **Conditional constant propagation**
  - Blocks: assume not executed until proven otherwise
  - Variables: assume not executed



- **Control-dependence**
  - Y is control-dependent on X if
    - X branches to u and v
    - $\exists$ path $u \rightsquigarrow exit$ which doesn't go through Y
    - $\forall$ paths $v \rightsquigarrow exit$ go through Y



X can determine whether Y is executed

- **Control Dependence Graph**
  1. Construct CFG
  2. Add entry and exit node
  3. Add (entry, exit) edge
  4. Create G' the reverse CFG
  5. Compute D-tree in G' (post-dominators of G)
  6. Compute $DF_{G'}(y)$ $\forall y \in G'$ (post-DF of G)
  7. Add $(x,y) \in G$ to CDG if $x \in DF_{G'}(y)$

- **Dead Code Elimination**

Dead Code Elimination

```
W ← list of all defs
while !W.isEmpty {
        Stmt S ← W.removeOne
        if |S.users| != 0 then continue
        if S.hasSideEffects() then continue
        foreach def in S.operands.definers {
          def.users ← def.users - {S}
          if |def.users| == 0 then
            W ← W UNION {def}
        }
        delete S
}
```

can leave **zombies**
"alive but useless"

Carnegie Mellon

Todd C. Mowry 15-745: SSA-Style Optimizations 15

Aggressive Dead Code Elimination (Fixed Version)

Assume a statement is dead until proven otherwise.

```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert:
       - S.operands.definers into W
       - S.CD^-1 into W

while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert:
       - S.operands.definers into W
       - S.CD^-1 into W
}
```

Conditional tests for blocks upon which S are control-dependent are also inserted into work-list.

Carnegie Mellon

Todd C. Mowry 15-745: SSA-Style Optimizations 23

Chapter 9.6 Loops in Flow Graphs

- Dominators

	Dominators
Domain	The power set of N
Direction	Forwards
Transfer function	$f_B(x) = x \cup \{B\}$
Boundary	$\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
Meet ($\wedge$)	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P, pred(B)} \text{OUT}[P]$
Initialization	$\text{OUT}[B] = N$

Figure 9.40: A data-flow algorithm for computing dominators

- Depth-First Ordering = reverse postorder
  - Constructing a depth-first spanning tree
    - Advancing edges $m \to$ proper descendant of m
    - Retreating edges $m \to$ ancestor of m (possibly itself)
    - Cross edges neither advancing nor retreating
      - If DFST drawn st children drawn L→R in the order in which they were added to the tree, then all cross edges R→L
- Back edges: edge $t \to h$ where h dom t
  - In a flow graph, every back edge retreating, but not every retreating is back
  - Flow graph reducible if all retreating edges in DFST are also back edges
    - Most tasteful programs are reducible
- Depth of DFST = largest number of retreating edges on any cycle-free path
  - Intuitively upper bound on loop nesting

- Natural Loops
  - Properties
    - Single-entry node (header)
    - $\exists$ back edge entering the loop header
  - Natural loop of a back edge $n \to d = \{d\} \cup \{v : v$ can reach n without going through $d\}$

- Convergence of iterative dataflow algorithms
  - Generally one+depth to carry variable use backward along any acyclic path

---

Lecture 2/24

- Motivation: uniform treatment for all loops
- Not every cycle is a loop from optimization perspective
- Loop
  - Single entry point (header): dominates all nodes in the loop
  - Edges must form at least a cycle
  - Loops can nest
- Back edge: arc $t \to h$ where h dominates t
  - Natural loop of back edge $t \to h$ is the smallest set of nodes that
    - Includes t and h
    - No predecessors outside of the set except for predecessors of header h

Lecture 2/24 cont.

- **Inner Loops**
  - Loops L_1 L_2 different header? → disjoint or one is inner loop
  - Loops L_1 L_2 same header? → combine and treat as one loop
- **Preheader**: optimization code to be executed before every loop

- **LICM** Loop Invariant Code Motion
  - **Loop-invariant computation**: computation whose value doesn't change as long as control stays within loop
  - **Code motion**: move statement within loop to preheader of loop

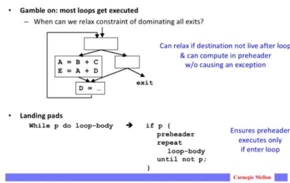
- **Finding loop-invariant computation**
  - Reaching definitions

  ① INVARIANT if all defs of B, C that reach A=B+C are outside loop

  ② Repeat until no new loop-invariant statements:

  INVARIANT if all reaching defs of B/C are outside loop, or exactly one reaching def for B/C from INVARIANT statement within the loop

- **Code Motion**
  - Conditions
    - Correctness : no change in semantics
    - Performance : no slowdown
    - **"Defines once and for all"**
      - Control flow once : code dominates all exits
      - Other defs for all: no other defs
      - Uses of def for all: dom use or no other reaching defs to use
  - **Aggressive Optimization**



- Induction variables and strength reduction (bonus material since schedule moved)
  - Just went over motivation, e.g., array access by index instead of pointer

---

Chapter 9.5 Partial Redundancy Elimination (9.5–9.2)

- Partial redundancy elimination: minimizing the number of expression evaluations
- Expression e is fully redundant at point p if it is an available expression at that point
- Not all redundancy can be eliminated unless we can change the flow graph
- Critical edge = any edge leading from a node with more than one successor to a node with more than one predecessor

Lecture 2/25

- Expression e **partially redundant** at P if E is partially available there (evaluated along at least one path to P)
- Can **insert computation** to make partially redundant fully redundant
- Loop invariants are partial redundancies
- **Partially available**

  0 ↓ 1 fwd, entry = 0, meet = ∪

  $PAV_{OUT} = (PAV_{IN} - KILL) \cup AVLOC$ (AVLOC: locally available ie downwards exposed)

  $PAV_{IN} = \begin{cases} 0 & \text{if entry} \\ \bigcup_{p \in pred} PAV_{OUT}(p) & \text{else} \end{cases}$

- **Anticipated expressions**

  1 ↓ 0 backward, exit = 0, meet = ∩

  $ANT_{IN} = ANTLOC \cup (ANTOUT - KILL)$ (ANTLOC: locally anticipated ie upwards exposed)

  $ANT_{OUT} = \begin{cases} 0 & \text{if exit} \\ \bigcap_{s \in succ} ANT_{IN} & \text{else} \end{cases}$

- **Placement Possible**
  - Insert at earliest place where PP=1
  - PP_{IN} = Placement Possible or not necessary in each predecessor block
  - PP_{OUT} = Placement Possible at exit of block or before
  - Don't insert where already available $INSERT = PP_{OUT} \cap (\neg PP_{IN} \cup KILL) \cap \neg AV_{OUT}$
  - Remove upward-exposed where PP=1 $DELETE = PP_{IN} \cap ANTLOC$

Formulating the Problem

- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors
- **PPIN: we want to place at input of this block only if (all of):**
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we can gain something by placing it here (PAVIN)
- **Forward or Backward?**
  - **BOTH!**
- **Problem is *bidirectional*, but lattice {0, 1} is finite, so**
  - as long as transfer functions are monotone, it converges.

"Placement Possible" Correctness

- **Convergence of analysis: transfer functions are monotone.**
- **Safety: Insert only if anticipated.**

$$PPIN[i] \subseteq (PPOUT[i] - KILL[i]) \cup ANTLOC[i]$$

$$PPOUT[i] = \begin{cases} 0 & i = exit \\ \bigcap_{s \in succ(i)} PPIN[s] & otherwise \end{cases}$$

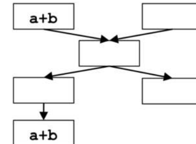
  - INSERT ⊆ PPOUT ⊆ ANTOUT, so insertion is safe.
- **Performance: never increase the # of computations on any path**
  - DELETE = PPIN ∩ ANTLOC
  - On every path from an INSERT, there is a DELETE.
  - The number of computations on a path does not increase.

Computing "Placement Possible"

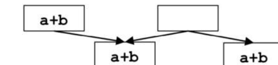
- **PPOUT: we want to place at output of this block only if**
  - we want to place at entry of all successors
  - $PPOUT[i] = \begin{cases} 0 & i = entry \\ \bigcap_{s \in succ(i)} PPIN[s] & otherwise \end{cases}$
- **PPIN: we want to place at start of this block only if (all of):**
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we gain something by moving it up (PAVIN heuristic)
  - $PPIN[i] = \begin{cases} 0 & i = exit \\ ([ANTLOC[i] \cup (PPOUT[i] - KILL[i])] \cap \bigcap_{p \in preds(i)} (PPOUT[p] \cup AVOUT[p]) \cap PAVIN[i]) & otherwise \end{cases}$

Morel-Renvoise Limitations

- **Movement usefulness tied to PAVIN heuristic**
  - Makes some useless moves, might increase register lifetimes:



  - Doesn't find some eliminations:



- **Bidirectional data flow difficult to compute.**

Related Work

- **Don't need heuristic**
  - Dhamdhere, Drechsler-Stadel, Knoop,et.al.
  - use restricted flow graph or allow edge placements.
- **Data flow can be separated into unidirectional passes**
  - Dhamdhere, Knoop, et. al.
- **Improvement still tied to accuracy of computational model**
  - Assumes performance depends only on the number of computations along any path.
  - Ignores resource constraint issues: register allocation, etc.
  - Knoop, et.al. give "earliest" and "latest" placement algorithms which begin to address this.
- **Further issues:**
  - more than one expression at once, strength reduction, redundant assignments, redundant stores.

Chapter 9.5.3 Lazy Code Motion to 9.5.5

- Lazy Code Motion
  - Eliminating partial redundancy with the goal of delaying computations as much as possible
  - Minimize register lifetimes

	(a) Anticipated Expressions	(b) Available Expressions
Domain	Sets of expressions	Sets of expressions
Direction	Backwards	Forwards
Transfer function	$f_B(x) = e_use_B \cup (x - e_kill_B)$	$f_B(x) = (anticipated[B].in \cup x) - e_kill_B$
Boundary	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cap$	$\cap$
Equations	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialization	$\text{IN}[B] = U$	$\text{OUT}[B] = U$

	(c) Postponable Expressions	(d) Used Expressions
Domain	Sets of expressions	Sets of expressions
Direction	Forwards	Backwards
Transfer function	$f_B(x) = (earliest[B] \cup x) - e_use_B$	$f_B(x) = (e_use_B \cup x) - latest[B]$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$
Meet ($\wedge$)	$\cap$	$\cup$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$
Initialization	$\text{OUT}[B] = U$	$\text{IN}[B] = \emptyset$

$$earliest[B] = anticipated[B].in - available[B].in$$
$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \Big(e_use_B \cup \neg\big(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S].in)\big)\Big)$$

Figure 9.34: Four data-flow passes in partial-redundancy elimination

Putting it All Together

All the steps of the algorithm are summarized in Algorithm 9.36.

Algorithm 9.36: Lazy code motion.

INPUT: A flow graph for which e_use_B and e_kill_B have been computed for each block B.

OUTPUT: A modified flow graph satisfying the four lazy code motion conditions in Section 9.5.3.

METHOD:

1. Insert an empty block along all edges entering a block with more than one predecessor.
2. Find $anticipated[B].in$ for all blocks B, as defined in Fig. 9.34(a).
3. Find $available[B].in$ for all blocks B as defined in Fig. 9.34(b).
4. Compute the earliest placements for all blocks B:
$$earliest[B] = anticipated[B].in - available[B].in$$
5. Find $postponable[B].in$ for all blocks B as defined in Fig. 9.34(c).
6. Compute the latest placements for all blocks B:
$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \Big(e_use_B \cup \neg\big(\bigcap_{S \text{ in } succ(B)} (earliest[S] \cup postponable[S].in)\big)\Big)$$
Note that $\neg$ denotes complementation with respect to the set of all expressions computed by the program.
7. Find $used[B].out$ for all blocks B, as defined in Fig. 9.34(d).
8. For each expression, say $x+y$, computed by the program, do the following:
   (a) Create a new temporary, say t, for $x + y$.
   (b) For all blocks B such that $x + y$ is in $latest[B] \cap used[B].out$, add `t = x+y` at the beginning of B.
   (c) For all blocks B such that $x + y$ is in
   $$e_use_B \cap (\neg latest[B] \cup used.out[B])$$
   replace every original $x + y$ by t.

□

Lecture 3/2

- Lazy code motion
  - Replace bi-directional dataflow Placement Possible with 4 separate unidirectional dataflow problem
  - Big picture
    - Earliest: maximize redundancy elimination, but long register lifetimes
    - Latest : same amount of redundancy elimination, but shorter register lifetimes

- Critical edges
  - Source has multiple successors
  - Destination has multiple predecessors

- Full redundancy = cut set (nodes separating entry from p, containing calculation)
- Partial redundancy = completing a cut set by adding operations
- The rest of lecture seems to be going into lazy code motion weeds

Chapter 9.7 Region-Based Analysis

- Previously: iterative dataflow analysis
  1. Create transfer function for basic blocks
  2. Find fixedpoint solution by repeated passes over the blocks
- Now: region-based analysis
  - Find transfer functions that summarize the execution of progressively larger regions of the program
  - Goal: transfer functions for entire procedures/programs
  - Better summarizes the effect of loops

- **Region-based analysis**
  - Program = hierarchy of regions ≈ portions of flow graph that have only one point of entry
  - Region = (nodes N, edges E) where
    - $\exists$ header h in N that dominates all nodes in N
    - If $\exists m \overset{\notin h}{\rightsquigarrow} n$, $n \in N$, then $m \in N$
    - E is the set of all control flow edges between any $n_1, n_2 \in N$, except possibly for some edges entering h
  - Focus on forward flow, backward flow is complicated

- **Constructing region hiearchy**
  - Assuming reducible flow graph, otherwise perform node splitting first (details later)
  - Algorithm
    1. Every block is a region by itself (leaf region)
    2. for each natural loop in inside-out (innermost first):
       - to replace loop L with a single node,
       1. Replace the body of L by region R

          $\left\{ \begin{matrix} \cdot \rightarrow \text{header}(L) \\ \text{exit}(L) \rightarrow \cdot \\ L \xrightarrow{\text{back edge}} L \end{matrix} \right\} \rightarrow \left\{ \begin{matrix} \cdot \rightarrow R \\ R \rightarrow \cdot \\ R \circlearrowleft \end{matrix} \right\}$

          R is called a body region
       2. Construct region R' representing entire natural loop L
          - R' is called a loop region
          - R' = R + back edges to header of loop L
    3. If the entire flow graph is not a natural loop, add the region consisting of the entire flow graph

- **Reducible flow graphs** = all graphs reducible to a single node by T_1 and T_2 rules
  - T_1: remove an edge from a node to itself
  - T_2: if node n has single predecessor m, and $n \neq$ flow graph entry, combine m and n

- **Necessary transfer function assumptions**
  - Composition was all that iterative dataflow needed
  - Meet: $(f_1 \wedge f_2)(x) = f_1(x) \wedge f_2(x)$
  - Closure: $f^* = \bigwedge_{n \geq 0} f^n$ where f^n = going around the cycle n times

Chapter 9.7 cont.

- Algorithm

Algorithm 9.53: Region-based analysis.

INPUT: A data-flow framework with the properties outlined in Section 9.7.4 and a reducible flow graph G.

OUTPUT: Data-flow values $\text{IN}[B]$ for each block B of G.

METHOD:

1. Use Algorithm 9.52 to construct the bottom-up sequence of regions of G, say $R_1, R_2, \ldots, R_n$, where R_n is the topmost region.
2. Perform the bottom-up analysis to compute the transfer functions summarizing the effect of executing a region. For each region $R_1, R_2, \ldots, R_n$, in the bottom-up order, do the following:
   (a) If R is a leaf region corresponding to block B, let $f_{R,\text{IN}[B]} = I$, and $f_{R,\text{OUT}[B]} = f_B$, the transfer function associated with block B.
   (b) If R is a body region, perform the computation of Fig. 9.50(a).
   (c) If R is a loop region, perform the computation of Fig. 9.50(b).
3. Perform the top-down pass to find the data-flow values at the beginning of each region.
   (a) $\text{IN}[R_n] = \text{IN}[\text{ENTRY}]$.
   (b) For each region R in $\{R_1, \ldots R_{n-1}\}$, in the top-down order, compute $\text{IN}[R] = f_{R',\text{IN}[R]}(\text{IN}[R'])$, where R' is the immediate enclosing region of R.

Algorithm 9.52: Constructing a bottom-up order of regions of a reducible flow graph.

INPUT: A reducible flow graph G.

OUTPUT: A list of regions of G that can be used in region-based data-flow problems.

METHOD:

1. Begin the list with all the leaf regions consisting of single blocks of G, in any order.
2. Repeatedly choose a natural loop L such that if there are any natural loops contained within L, then these loops have had their body and loop regions added to the list already. Add first the region consisting of the body of L (i.e., L without the back edges to the header of L), and then the loop region of L.
3. If the entire flow graph is not itself a natural loop, add at the end of the list the region consisting of the entire flow graph.

□

```
1)  for (each subregion S immediately contained in R, in
                                      topological order) {
2)      f_{R,IN[S]} = ∧_{predecessors B in R of the header of S} f_{R,OUT[B]};
        /* if S is the header of region R, then f_{R,IN[S]} is the
              meet over nothing, which is the identity function */
3)      for (each exit block B in S)
4)          f_{R,OUT[B]} = f_{S,OUT[B]} ∘ f_{R,IN[S]};
    }
```

(a) Constructing transfer functions for a body region R

```
1)  let S be the body region immediately nested within R; that is,
        S is R without back edges from R to the header of R;
2)  f_{R,IN[S]} = (∧_{predecessors B in R of the header of S} f_{S,OUT[B]})*;
3)  for (each exit block B in R)
4)      f_{R,OUT[B]} = f_{S,OUT[B]} ∘ f_{R,IN[S]};
```

(b) Constructing transfer functions for a loop region R'

Figure 9.50: Details of region-based data-flow computations

- Node-splitting
  - Handles nonreducible flow graphs

We pick some region R that has more than one predecessor and is not the header of the entire flow graph. If R has k predecessors, make k copies of the entire flow graph R, and connect each predecessor of R's header to a different copy of R. Remember that only the header of a region could possibly have a predecessor outside that region. It turns out, although we shall not prove it, that such node splitting results in a reduction by at least one in the number of regions, after new back edges are identified and their regions constructed. The resulting graph may still not be reducible, but by alternating a splitting phase with a phase where new natural loops are identified and collapsed to regions, we eventually are left with a single region; i.e., the flow graph has been reduced.

We must also think about how the result of the data-flow analysis on the split flow graph relates to the answer we desire for the original flow graph. There are two approaches we might consider.

1. Splitting regions may be beneficial for the optimization process, and we can simply revise the flow graph to have copies of certain blocks. Since each duplicated block is entered along only a subset of the paths that reached the original, the data-flow values at these duplicated blocks will tend to contain more specific information than was available at the original. For instance, fewer definitions may reach each of the duplicated blocks that reach the original block.
2. If we wish to retain the original flow graph, with no splitting, then after analyzing the split flow graph, we look at each split block B, and its corresponding set of blocks $B_1, B_2, \ldots, B_k$. We may compute $\text{IN}[B] = \text{IN}[B_1] \wedge \text{IN}[B_2] \wedge \cdots \wedge \text{IN}[B_k]$, and similarly for the OUT's.

Lecture 3/3

- Region-based analysis in practice
  - Faster for "harder" analysis
  - Useful for analyses related to structure

Optimization

- **Let m = number of edges, n = number of nodes**
- **Ideas for optimization**
  - If we compute $F_{R,B}$ for every region B is in, then it is very expensive
  - We are ultimately only interested in the entire region (E); we need to compute only $F_{E,B}$ for every B.
    - There are many common subexpressions between $F_{E,B1}$, $F_{E,B2}$, ...
    - Number of $F_{E,B}$ calculated = m
  - Also, we need to compute $F_{R,in(R')}$, where R' represents the region whose header is subsumed.
    - Number of $F_{R,B}$ calculated, where R is not final = n
- Total number of $F_{R,B}$ calculated: (m + n)
  - Data structure keeps "header" relationship
    - Practical algorithm: O(m log n)
    - Complexity: $O(m\alpha(m,n))$, α is inverse Ackermann function

Comparison with Iterative Data Flow Analysis

- **Applicability**
  - Definitions of F* can make technique more powerful than iterative algorithms
  - Backward flow: reverse graph is not typically reducible.
    - Requires more effort to adapt to backward flow than iterative algorithm
  - More important for interprocedural optimization, optimizations related to loop nesting structure
- **Speed**
  - Irreducible graphs
    - Iterative algorithm can process irreducible parts uniformly
    - Serious "irreducibility" can be slow with region-based analysis
  - Reducible graph & Cycles do not add information (common)
    - Iterative: (depth + 2) passes, O(m*depth) steps
      depth is 2.75 average, independent of code length
    - Region-based analysis: Theoretically almost linear, typically O(m log n) steps
  - Reducible graph & Cycles add information*
    - Iterative takes longer to converge
    - Region-based analysis remains the same

*E.g., Constant Propagation

L: a = b
b = c
c = 1
goto L

Chapter 12.4 Pointer Analysis + 12.6–12.7

- **Pointer aliasing**
  - If two pointers can point to the same object, then the pointers may be **aliased**
  - Difficult with arbitrary casting ((void*) 42), indirect function calls (virtual methods)

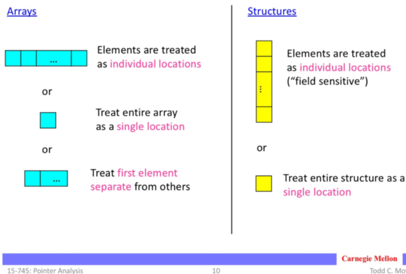
- **Points-to Analysis**
  - Simplify for now: **flow-insensitive, context-insensitive**
  - **Java program model**
    - Variables: refers to static/live on run-time stack variables of type pointer to T or reference to T
    - Heap objects: a heap of objects exist, all variables only point to heap objects not other variables
    - Fields: a heap object can have fields, the value of a field can be a reference to a heap object but not to a variable
  - **Flow-insensitive**
    - Assert variable v can point to heap object h
    - Can ignore: "where can v→h", "in what context can v→h"
    - Note: heap objects unnamed, "v $\xrightarrow{can}$ h" = "v can point to ≥1 of the objects created at statement h"
  - **Points-to analysis**: determine what each variable and each field of each heap object can point to
    - Two pointers are aliased if their points-to sets intersect
    - Different approaches
      - **Inclusion-based**: v=w causes v to point to all objects w points to, not vice versa
      - **Equivalence-based**: v=w turns v and w into one equivalence class
  - **Flow-insensitive**
    - Ignore control flow, statements can execute in any order
    - Assignment cannot kill, only generate
    - Reduce size of result representation and converge faster, but much weaker analysis
  - **Context-insensitive**
    - Not in reading
    - Parameters and returned values modeled by copy statements
    - Fancy datalog stuff
  - **Context-sensitive**
    - Problem: large summaries, exponentially many contexts
    - **Cloning-based analysis**
      - Clone the methods, one for each context of interest
      - Apply context-insensitive analysis to cloned call graph
      - But not uncommon to have $>10^{14}$ contexts in a Java application
    - **Two core problems**
      - Handling context sensitivity? Apply context-insensitive algorithm to cloned call graph
      - Represent exponentially many contexts? Use binary decision diagrams (BDDs)
    - Steps
      1. Run context-insensitive points-to to get a call graph
      2. Create a cloned call graph
         - Context = representation of call string forming history of active function calls, some yak shaving around recursion
         - BDD for context representation, but finicky (e.g., variable ordering)
      3. More datalog, see book. There are some mildly interesting optimizations

Lecture 3/4 Pointer Analysis

- **Representation**
  - Track pointer aliases : more precise, less efficient
  - Track points-to information : less precise, more efficient

- **Heap modeling**
  - Heap merged ("no heap modeling")
  - Allocation site (any call to malloc - each a unique location)
  - Shape analysis (trees, linked lists, etc)

- **Aggregate modeling**



- Flow-sensitivity
  - Flow insensitive - actually used in practice! very cheap
  - Flow sensitive - consider program points in CFG
  - Path sensitive - consider paths in CFG

- **Address taken**
  - Basic, fast, ultra-conservative O(n) analysis, very imprecise
  - Flow-insensitive, context-insensitive
  - Generate set of all variables whose addresses are assigned to another variable
  - Assume any pointer can point to any variable in that set

- **Andersen's Algorithm**
  - Flow-insensitive, context-insensitive, iterative
  - One points-to graph for entire program, each node represents exactly one location
  - To build the graph,

y = &x	y points-to x
y = x	if x points-to w then y points-to w
*y = x	if y points-to z and x points-to w then z points-to w
y = *x	if x points-to z and z points-to w then y points-to w

  - Iterate until graph no longer changes, $O(n^2)$

- **Steensgaard's Algorithm**
  - Flow-insensitive, context-insensitive
  - Compact (but less precise) points-to graph with union-find, each node can represent multiple locations but can only point to at most one other node, O(n)

Lecture 3/4 cont.

- **Binary decision diagrams**
  - Use BDD for representing transfer functions
  - Accurate and scales to large programs
  - Context-sensitive, inter-procedural analysis

- **Probabilistic pointer analysis**
  - Speculate with verify and recover
  - Can attempt to quantify benefits

- **Algorithm design**
  - Fixed
    - Top-down vs bottom up
    - Linear transfer functions
    - One-level context and flow sensitive
  - Flexible
    - Edge profiling vs static prediction
    - Safe vs unsafe
    - Field-sensitive vs field-insensitive

Chapter 8.8 Register Allocation

- Speed: Registers ≫ Memory ⋙ Disk
- **Register allocation**: what values should reside in registers?
- **Register assignment**: in which register should a value reside?

- **Global register allocation**
  - Keep registers consistent across block boundaries (globally) to save on some stores/loads
  - Assign some fixed number of registers to hold most active values in each inner loop

- **Usage counts**
  - $\text{Benefit} \approx \sum_{B \in L} use(x,B) + 2 * live(x,B)$
    - B: block; L: loop
    - $use(x,B)$: # times x used in B prior to def
    - $live(x,B)$: 1 if x live on exit from B and assigned value in B; 0 otherwise

- **Register spilling**
  - When all registers used, one must be spilled to memory
  - Two-pass graph coloring
    1. Assume ∞ symbolic registers
    2. Assign physical registers to symbolic ones
  - Although graph coloring NP-hard, good heuristic in practice:

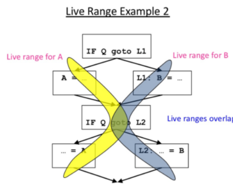
    if node n has <k neighbors
        remove n and its edges

    either:
    1. obtain empty graph, produce k-coloring by going in reverse
    2. not empty graph, use Chaitin's heuristics for spilling

Lecture 3/9 Register Allocation

- **Interference graph**
  - Undirected graph
  - Node = pseudo-register
  - Edge (n_1, n_2) if pseudo-registers n_1 and n_2 **interfere**, i.e., at some point in the program they cannot both occupy the same register

- **Live ranges**
  - Motivation: create an interference graph that is easier to color
  - Live range = live variables + reaching definitions
    - A live range is a definition and all program points in which that definition is live
  - Two overlapping live ranges **must be merged** (merged live ranges also known as webs)
    - Merging ≈ unconvert out of SSA
    - Optimization that is both faster and sometimes better; only check for interference at start of each live range



A and B can use same register!

- Graph coloring extended

IV. Extending Coloring: Design Principles

- A pseudo-register is
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- Objective function
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- Heuristics
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

15-745: Register Allocation 19 Carnegie Mellon Todd C. Mowry

Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

15-745: Register Allocation 20 Carnegie Mellon Todd C. Mowry

Chaitin: Coloring and Spilling

- Apply coloring heuristic

```
Build interference graph
Iterate until there are no nodes left
    If there exists a node v with less than n neighbor
        push v on register allocation stack
    else
        v = node with highest degree-to-cost ratio
        mark v as spilled
    remove v and its edges from graph
```

- Spilling may require use of registers (must reload at each use, store at each def); change interference graph

```
While there is spilling
    rebuild interference graph and perform step above
```

- Assign registers

```
While stack is not empty
    Pop v from stack
    Reinsert v and its edges into the graph
    Assign v a color that differs from all its neighbors
```

15-745: Register Allocation 21 Carnegie Mellon Todd C. Mowry

Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?
- One Heuristic:
  - Cost-to-degree-ratio = $[(\# \text{ defs \& uses}) * 10^{\text{loop-nest-depth}}]/\text{degree}$
  - Spill node with highest degree-to-cost ratio

15-745: Register Allocation 22 Carnegie Mellon Todd C. Mowry

Quality of Chaitin's Algorithm

- Problem: Can give up on coloring too quickly

n=2 (graph: B, E, A, C, D)

No spilling required, but Chaitin's Algorithm spills

An optimization: "Prioritize the coloring"
- Still eliminate a node and its edges from graph
- Do not commit to "spilling" just yet
- Try to color again in assignment phase

- Problem: All or nothing
  - Why not try to keep a pseudo-register in a hardware register part of the time?

15-745: Register Allocation 23 Carnegie Mellon Todd C. Mowry

Splitting Live Ranges

- Different perspective: Instead of choosing variables to spill, choose live ranges tc
- Split pseudo-registers into live ranges to make interference graph easier to color
  - Eliminate interference in a variable's "dead" zones
  - Increase flexibility in allocation:
    - can allocate same variable to different registers

```
A =
IF A goto L1

B =          L1: C =
  = A              = A
D =            D =
  = B+D            = C+D

A = D

  = A
```

n=2 (graph: A1, B, C, D, A2)

15-745: Register Allocation 24 Carnegie Mellon Todd C.

Insight

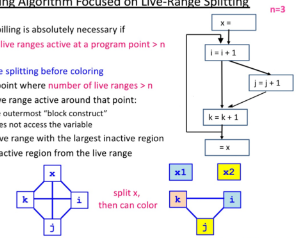
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's "nearly dead" zones
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - Initially: # active live ranges at a program point can be > # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - Goal: # active live ranges cannot be > # registers

15-745: Register Allocation 25 Carnegie Mellon Todd C. Mowry

A Spilling Algorithm Focused on Live-Range Splitting

Observation: spilling is absolutely necessary if
- number of live ranges active at a program point > n

Apply live-range splitting before coloring
- Identify a point where number of live ranges > n
- For each live range active around that point:
  - find the outermost "block construct" that does not access the variable
- Choose a live range with the largest inactive region
- Split the inactive region from the live range



15-745: Register Allocation 29 Carnegie Mellon Todd C. Mowry

- Should probably writeup ↑ at some point
- Also mentioned: coalescing

Chapter 10.1 Processor Architectures + 10.2

- **Instruction pipelining**
  - Goal: instruction-level parallelism
  - Branch instructions problematic
    - Many processors: speculatively fetch and decode immediately succeeding branch not taken instructions
    - When branch taken, empty instruction pipeline, fetch branch target
    - Advanced processors use hardware to predict branches based on execution history
- **Pipelined execution**
  - Some instructions take several clocks to execute, e.g., memory load
  - An instruction's execution is **pipelined** if succeeding instructions not dependent on the result are allowed to proceed
  - Most general purpose processors dynamically detect dependencies between consecutive instructions, automatically stall
  - Simple/low-power processors (e.g. embedded) require compiler to insert no-ops
- Managing parallelism
  - Software: **VLIW** (very long instruction word) machines
    - Wider instruction words encode the operations to be issued in a single clock
  - Hardware: **superscalar**
    - Automatically detect dependencies between instructions, issue as operands become available
  - Some processors have both VLIW and superscalar functionality
- **Hardware schedulers**
  - Simple: execute instructions in order of fetch
  - Sophisticated: execute **out of order,** buffer stalled operations
  - Both benefit from static scheduling

- Code scheduling constraints
  - **Control-dependence**: all operations executed in original must be executed in optimized version
  - **Data-dependence**: operations in optimized must produce same results as operations in original
  - **Resource constraints**: schedule must not oversubscribe the resources on the machine
  - Guarantees **same result,** but **not same memory states** - harder to debug

- Data-dependence types
  - **True dependence**: RAW read-after-write
  - **Antidependence**: WAR write-after-read } storage-related dependencies, can eliminate by using different locations to store different values
  - **Output dependence**: WAW write-after-write }
  - Data dependencies apply to both memory and register access

- Finding dependencies
  - Generally undecidable at compile-time
  - Highly sensitive to programming language used

- **Register Usage vs Parallelism**
  - Traditional register allocation minimizes number of registers used
  - But using the same register introduces storage dependencies
  - Computer architects introduced **hardware register renaming** to undo this: dynamically change register assignment

Chapter 10.2 cont.

- **Phase ordering**
  - Register allocation → scheduling: many storage dependencies
  - Scheduling → register allocation: may require so many registers that excessive register spilling occurs
  - Depends on characteristics of the program being compiled

- **Control dependence**
  - Basic blocks small on average (<5 instructions)
  - Operations within same block often highly related, little parallelism
  - Instruction i_1 control-dependent on instruction i_2 if the output of i_2 determines whether i_1 executed

    if (cond) { s_1; } else { s_2; }

    s_1 and s_2 are control-dependent on cond
  - Can speedup program with speculative execution

- **Speculative execution support** in processors
  - Prefetch
  - Poison bits
  - Predicated execution

- Skipping over **machine model**

Lecture 3/10 Local Instruction Scheduling

- Why not make deeper and deeper pipelines?
  - In between stages, CPU registers must be stored. Diminishing returns, Amdahl's Law.
  - Pipeline stage value unclear if already faster than integer add

- **Scheduling limitations**
  - **Hardware resources**
    - Finite issue width
    - Limited functional units for each instruction type
    - Limited pipelining within one functional unit
  - **Data dependencies**
    - Some instructions take more than one cycle to execute
  - **Control dependencies**

- **List Scheduling**: within a basic block
- **Global Scheduling**: across basic blocks
- **Software Pipelining**: across loop iterations

Lecture 3/10 cont.

- **List Scheduling**
  - **NP-hard**
  - Input
    - **Data precedence graph**
      - True edges E RAW dependency, must wait for completion to start next
      - Anti edges E' WAR dependency, can start together as long as next finishes later
    - **Machine parameters** (# FUs, latencies)
  - Output
    - **Scheduled code** (which instructions to start in a cycle)
  - **Algorithm**
    - Maintain a list of instructions that are ready to execute
    - Moving cycle-by-cycle through the schedule template,
      - Choose instructions from the list and schedule them based on priorities
      - Update the list for the next cycle
  - **Priorities**
    - Factors
      - Data dependencies
      - Machine parameters, e.g., latencies
    - For true dependencies only
      - Priority = **latency-weighted depth** in DAG
      - $\text{Priority}(x) = \max\left(\forall_{\ell \in leaves(DPG)} \forall_{p \in paths(x, \dots, \ell)} \sum_{p_i = x}^{\ell} \text{latency}(p_i)\right)$
    - To account for anti-dependencies
      - Priority(x)

$$= \begin{cases} \text{latency}(x) + \max\left(\text{latency}(x) + \max_{(x,y) \in E} \text{priority}(y)\right) & \text{if } x \text{ is a leaf} \\ \max_{(x,y) \in E'} \text{priority}(y) & \text{otherwise} \end{cases}$$

List Scheduling Algorithm

```
cycle = 0;
ready-list = root nodes in DPG;
inflight-list = {};

while (|ready-list|+|inflight-list| > 0) {        <- ties?
   for op = (all nodes in ready-list in decreasing priority order) {
      if (an FU exists for op to start at cycle) {
         remove op from ready-list and add to inflight-list;
         add op to schedule at time cycle;
         if (op has an outgoing anti-edge)
            add all targets of op's anti-edges that are ready to ready-list;
      }
   }
   cycle = cycle + 1;
   for op = (all nodes in inflight-list)
      if (op finishes at time cycle) {
         remove op from inflight-list;
         check nodes waiting for op &
                            add to ready-list if all operands available;
      }
   }
}
```

Carnegie Mellon

15745: Instruction Scheduling 46

- **Backward List Scheduling**
  - Reverse direction of all DPG edges
  - Schedule the finish times of each operation
    - Though start times still needed for FU
  - Will **cluster operations near the end** instead of near the beginning
  - May be better/worse than forward scheduling

- **Evaluation**
  - **RBF scheduling**
    - Schedule each block M times backward and forward
    - Break ties randomly
  - **For real programs, regular list scheduling works very well**
  - For synthetic blocks, RBF wins when available parallelism is ~2.5
    - <2.5, scheduling too constrained
    - >2.5, any decision tends to work well
  - **List scheduling widely used on in-order processors**

Lecture 3/11 Global Scheduling and Software Pipelining

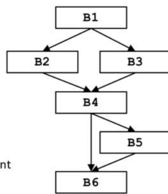
should read 10.3–10.5

- Control-related terminology
  - Two operations o_1 and o_2
    - **Control-equivalent** if o_1 is executed iff o_2 is executed
    - o_2 **control-dependent** on o_1 iff execution of o_2 dependent on outcome of o_1
  - Operation o is **speculatively executed** if it is executed before all the operands it depends on control-wise have been executed
    - But cannot raise an exception
    - And must satisfy data dependencies

- **Basic global scheduling**
  - Schedule innermost loops first
  - Only upward code motion to either
    - Control-equivalent block (non-speculative)
    - Control-equivalent block of a dominating predecessor (speculative, 1 branch)
  - No creation of copies

Useful Definitions

- Blocks B and B′ are control equivalent if
  - B is executed if and only if B′ is executed
  - E.g., which sets of blocks are control equivalent?



Note: Two ops (instructions) are control equivalent iff their basic blocks are control equivalent (could be from same basic block)

- NonSpeculative(B) = all blocks that are control equivalent to B and dominated by B
- Speculative(B) = all blocks B′ not control equivalent to B such that
  - B′ is a successor of at least one block B′′ that is control equivalent to B, and
  - B′ is dominated by B′′

NonSpeculative(B1)?
NonSpeculative(B2)?
Speculative(B1) ?
Speculative(B2) ?

Move up to a control-equivalent block or a control-equivalent block of a dominating predecessor

Carnegie Mellon

15-745: Global Scheduling 9

Basic Algorithm

```
Compute data dependences;
For each region R in the hierarchy of loop regions from inner to outer {
  For each basic block B of R in prioritized topological order {
    CandInsts = ready instructions in NonSpeculative(B) ∪ Speculative(B);
    For (t = 1, 2, ... until all instructions from B are scheduled) {   // schedule time slots in order
      For (n in CandInst in priority order) {                         // may or may not be from B
        if (ok to move n to B && n has no resource conflicts at time t) {
          S(n) = < B, t > ;                  // instruction n is mapped to basic block B and time slot t
          Update resource commitments;
          Update data dependences;
        }
      }
      Update CandInsts;                      // scheduled insts will often make new insts ready
    }
  }
}
```

Priority functions: Non-speculative before speculative, and otherwise use same priority as in list schedulin
Ok to move: Don't speculatively move a store instruction, don't move a procedure call, etc

Carnegie Mellon

15-745: Global Scheduling 10

Extension

- **In region-based scheduling, loop iteration boundary limits code motion: operations from one iteration cannot overlap with those from another**
- **Prepass before scheduling: loop unrolling**
- **Especially important to move operation up loop back edges**

```
for (i = 0; i < N; i++) {
    S(i);
}
```

Original Loop

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

Unrolled Loop

Carnegie Mellon

15-745: Global Scheduling 15

- **Software pipelining**
  - Across loop iterations
  - Unlike loop unrolling, can give optimal result with small code size blowup
  - Goals
    - Maximize throughput
    - Small code size
  - Find
    - An identical relative schedule $S(n)$ for every iteration
    - A constant initiation interval T
  - Such that
    - Initiation interval is minimized
  - NP-complete

- See textbook/slides
- Goes in the weeds
- Has inequalities etc

Algorithm for Acyclic Graphs

- **Find lower bound of initiation interval: T_0**
  - based on resource constraints
- **For $T = T_0, T_0+1, \ldots$ until all nodes are scheduled**
  - For each node n in topological order
    - s_0 = earliest n can be scheduled
    - for each $s = s_0, s_0+1, \ldots, s_0+T-1$
    - if NodeScheduled(n, s) break;
    - if n cannot be scheduled break;
- NodeScheduled(n, s)
  - Check resources of n at s in modulo resource reservation table
- Can always meet the lower bound if:
  - every operation uses only 1 resource, and
  - no cyclic dependences in the loop

Carnegie Mellon

15-745: Software Pipelining 29 Todd C. Mowry

Algorithm

- **Normally, every iteration uses the same set of registers**
  - introduces artificial anti-dependences for software pipelining
- **Modulo variable expansion algorithm**
  - schedule each iteration ignoring artificial constraints on registers
  - calculate life times of registers
  - degree of unrolling = max_r ($lifetime_r$ /T)
  - unroll the steady state of software pipelined loop to use different registers
- **Code generation**
  - generate one pipelined loop with only one exit (at beginning of steady state)
  - generate one unpipelined loop to handle the rest
  - code generation is the messiest part of the algorithm!

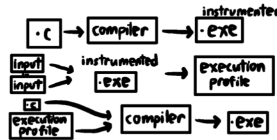
Carnegie Mellon

15-745: Software Pipelining 36 Todd C. Mowry

Lecture 3/16 Dynamic Code Optimization

- Motivation
  - Understanding common dynamic behaviors helps optimization (control flow, data dependencies, input values)
  - Useful for speculative scheduling, cache optimizations, code specializations, etc.

- **Profile-Based Compile-Time Optimization**
  1. Compile statically
  2. Collect execution profile
  3. Re-compile with execution profile



  - Collecting control flow profile usually not too expensive
  - Collecting input values profile much more expensive

- **Instrumenting executable binaries**
  - Compiler could insert instrumentation directly
  - **Binary instrumentation tool** could modify executable directly

- **Binary instrumentation approaches**
  - **Static binary-to-binary rewriting**
    - Challenges
      - Input is binary
      - Optimization: No source code, less info than original compiler
      - Instrumentation: Time/space overhead of instrumenting code
  - **Interpreter**
    - Grab, decode, emulate each instruction
    - Good: works for dynamic languages, easy to change execution on the fly
    - Bad: runtime overhead
  - **Sweet spot?**
    - Want: **flexibility** of interpreter, **performance** of direct hardware execution
    - Increase the **granularity** of interpretation (instructions → chunks of code)
    - **Dynamically compile** code chunks into directly-executed optimized code
      - Cache compiled chunks into software code cache
      - Jump in and out of cached chunks as appropriate
      - The cached chunks can be updated
    - Invest more time into optimizing code chunks that are clearly **hot**

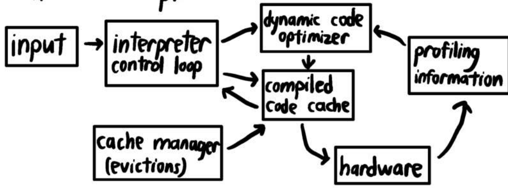
- **Chunk-Based Dynamic Optimizer**

```
while still_executing:
    if not compiled(pc):
        compileAndCache(pc)
    jumpIntoCache(pc)
    pc = getNextPC()
```

pc=program counter
also, not all code needs to be compiled (e.g., adaptive execution)

Lecture 3/16 cont.

- Typical JIT compiler



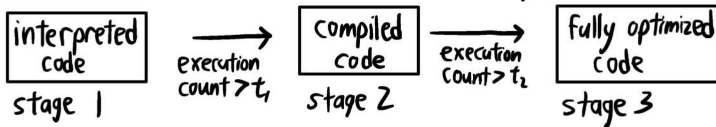
- **Dynamic Compilation Policy**

$$\Delta T_{total} = T_{compile} - (n_{executions} \cdot T_{improvement})$$

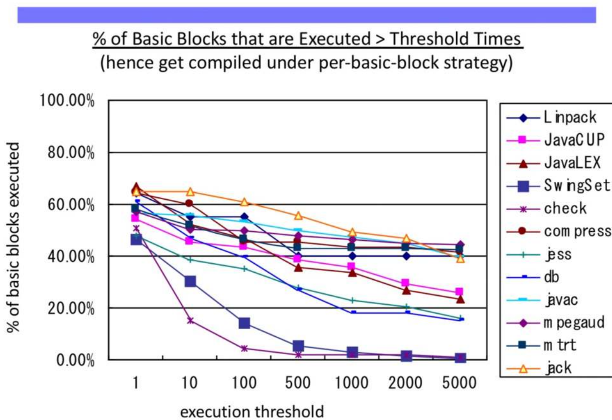
- ΔT_{total}: want this to be negative
- $T_{compile}$: faster compile times?
- $n_{executions}$: focus on hot spots
- $T_{improvement}$: better code but slower compilation?

	Startup Speed	Execution Performance
Interpreter	Best	Poor
Quick compiler	Fair	Fair
Optimizing compiler	Poor	Best

- **Multi-stage compilation**
  - **Execution count** = method invocations + back edges executed

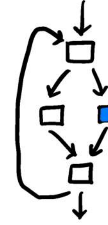


- **Compilation Granularity**
  - Usually not per method, even hot methods have rarely executed code
  - Compilation time $\propto$ amount of code compiled
  - Methods can be large especially with inling, but inlining very important for performance
  - Simple technique
    - Track execution counts for basic blocks in stages 1 and 2
    - Basic blocks that execute in stage 2 are not rare



Lecture 3/16 cont.

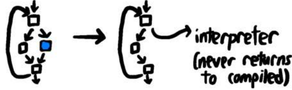
• Partial Method Compilation



Goal: white blocks compiled, blue block interpreted

New challenges:
- Transition white→blue, blue→white
- Compile/optimize ignoring blue

① From profile data, determine the set of rare blocks
② Determine live variables at rare block entry points
③ Redirect control flow for rare blocks



④ Perform compilation normally
⑤ Record a map for each interpreter transfer point
- Map: live variables → location (register/memory)
- Typically <100 bytes
- Used to reconstruct interpreter state

$$\begin{cases} x \to sp-4 \\ y \to r_1 \\ z \to sp-8 \end{cases}$$

• Partial Dead-Code Elimination

- Move computation that is only live on a rare path into the rare block
- May undo an optimization

```
x=0
if (UNLIKELY(cond1)){
   ...
}
if (UNLIKELY(cond2)){
   ...
}
```

→

```
if (UNLIKELY(cond1)){
   x=0
   ...
}
if (UNLIKELY(cond2)){
   x=0
   ...
}
```

• Escape Analysis

- Find objects that do not escape a method or a thread
- "Captured" by
  - Method: allocate on stack/in registers instead of heap, or scalar replacement (replace fields with local variables)
  - Thread: can avoid synchronization operations

• Partial Escape Analysis

- Stack allocate objects that don't escape in the common blocks
- Eliminate synchronization on objects that don't escape in the common blocks
- If ends up branching to a rare block:
  - Copy stack → heap, update pointers
  - Reapply eliminated synchronizations
- Examples
  - Graal

Benefits from Partial Escape Analysis

		MB / Iteration			MAllocs. / Iteration			Iterations / Minute		
		without	with	Δ	without	with	Δ	without	with	Speedup
DaCapo*	fop	172	166	-3.5%	3	3	-5.6%	150.75	172.41	+14.4%
	h2	1,336	1,267	-5.2%	31	30	-5.9%	11.64	11.98	+2.9%
	jython	2,242	2,057	-8.3%	28	23	-15.2%	25.35	24.80	-2.1%
	sunflow	2,707	2,010	-25.7%	62	43	-30.6%	54.55	55.40	+1.6%
	tomcat	691	685	-0.8%	7	7	-2.4%	46.73	48.78	+4.4%
	tradebeans	3,640	3,354	-7.8%	64	57	-11.1%	9.97	10.61	+6.4%
	xalan	1,289	1,270	-1.4%	10	10	-2.2%	156.25	159.15	+1.9%
	average†			-4.9%			-8.0%			+2.2%
ScalaDaCapo	actors	1,866	1,550	-17.0%	56	45	-18.5%	17.10	18.81	+10.0%
	apparat	3,418	3,306	-3.3%	74	70	-5.5%	6.11	6.94	+13.7%
	factorie	43,393	17,996	-58.5%	1,397	547	-60.9%	1.95	2.59	+33.0%
	kiama	642	600	-6.6%	13	11	-11.2%	116.28	135.44	+16.5%
	scalac	758	648	-14.5%	19	15	-22.6%	23.09	24.12	+4.4%
	scaladoc	1,189	1,046	-12.0%	24	18	-24.0%	20.39	20.99	+3.0%
	scalap	68	62	-8.8%	2	2	-12.5%	472.44	555.56	+17.6%
	scalariform	337	292	-13.3%	10	8	-16.5%	127.66	137.61	+7.8%
	scalatest	263	261	-1.0%	4	3	-2.4%	58.14	62.24	+7.1%
	scalaxb	226	212	-5.9%	4	3	-13.8%	100.50	105.26	+4.7%
	specs	588	362	-38.4%	12	3	-72.0%	35.03	36.43	+4.0%
	tmt	2,798	2,698	-3.6%	38	34	-12.2%	13.06	13.50	+3.3%
	average			-15.2%			-22.7%			+10.4%
SPECjbb2005‡		11,608	9,741	-16.1%	180	111	-38.1%	11.07	12.04	+8.7%

Table 1: Evaluation of size and number of allocations, and performance on (Scala)DaCapo and SPECjbb2005

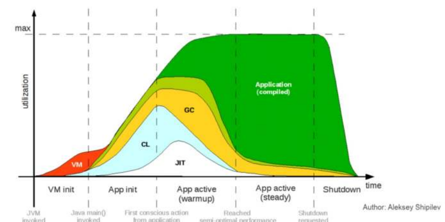
Carnegie Mellon
15-745: Dynamic Code Optimization 34 Todd C. Mowry

Dynamic Optimizations in HotSpot JVM

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph rep.
  - static single assignment rep.
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator

Carnegie Mellon
15-745: Dynamic Code Optimization 35 Todd C. Mowry

HotSpot JVM and Graal Dynamic Compiler



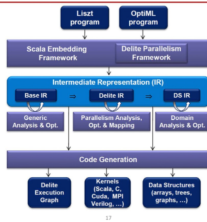
Carnegie Mellon
15-745: Dynamic Code Optimization 36 Todd C. Mowry

Lecture 3/17 Domain-Specific Languages

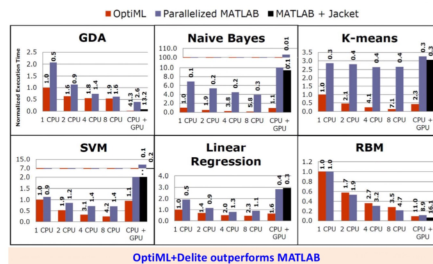
- MapReduce (open source version: Hadoop)
- GraphLab (think like a vertex)
- DSL Design Guidelines: [Karsai et al DSM09]

- Delite
  - Performance = heterogeneous + parallel
  - Goal: common DSL framework

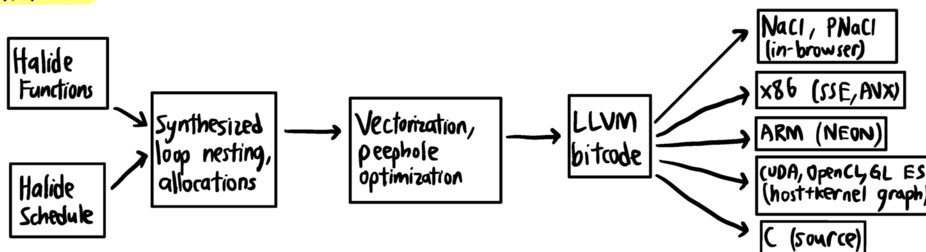
Delite DSL Compiler



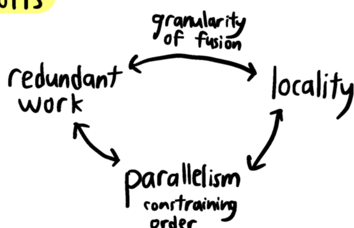
Experiments on ML kernels



- Halide: image processing
  - Designed for expert programmers
  - Systematic model for locality, parallelism, redundant computation in stencil pipelines
  - Scheduling representation to easily iterate – and an autotuner to empirically find good schedules
  - DSL compiler combines Halide programs and schedule descriptions
  - Loop synthesizer for data parallel pipelines based on simple interval analysis
    - Simpler and less expressive by polyhedral model
    - More general in class of analyzable expressions
  - Code generator for high quality vector code
  - The Halide talk slides are worth checking out
  - Halide: decouple algorithm from schedule
    - Algorithm: pipelines are pure functions from coordinates to values
    - Domain scope
      - Computations are over regular grids
      - Only feed-forward pipelines
      - Recursion must have bounded depth
    - Structure



- Tradeoffs



Lecture 3/18 Memory Hierarchy Optimizations

- Caches
  - Cache hierarchy
  - Typical configurations and parameters

- Optimizing caches
  - Temporal locality
  - Spatial locality
  - Minimize conflicts
- Time: reorder computation
  - When is an object accessed?
  - How to predict better access time?
  - How to ensure safety?
- Space: changing data layout
  - Where is an object located?
  - What are better layouts?
  - To what extent can layout be safely modified?

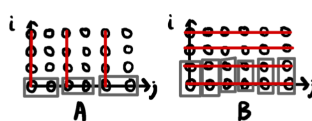
- Object Types
  - Scalars
  - Structures and pointers
  - Arrays

- Scalars
  - Locals
  - Globals
  - Procedure arguments
- Structures and pointers
  - Within node?
  - Across nodes?
- Arrays
  - Usually accessed within loop nests ⇒ easy to understand time
  - Understanding accesses: start of array, relative position in array

- Iteration space: each position represents one iteration, not the same as data space

```
for i=0 to N-1:
  for j=0 to N-1:
    A[i][j] = B[j][i]
```

if cache can hold 2 things



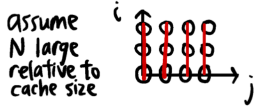
- Optimizing array accesses
  - When do cache misses occur? Locality analysis
  - Can change iteration order/data layout?
    - Evaluating cost
    - Checking correctness: dependence analysis

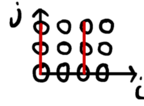
Lecture 3/18 cont.

- Some optimizations
  - **Loop Interchange**

```
for i=0 to N-1                for j=0 to N-1
  for j=0 to N-1        →       for i=0 to N-1
    A[j][i] = i*j                  A[j][i] = i*j
```

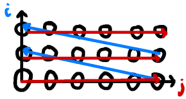
assume N large relative to cache size

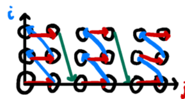




  - **Cache Blocking** (aka tiling)

```
for i=0 to N-1:                 for JJ=0 to N-1 by B:
  for j=0 to N-1:                 for i=0 to N-1:
    f(A[i], A[j])          →        for j=JJ to max(N-1, JJ+B-1):
                                      f(A[i], A[j])
```





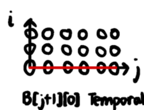
    - Can also be done in multiple dimensions, e.g., matrix multiply

- **Locality Analysis**
  - **Reuse**: accessing a location that has been accessed in the past
  - **Locality**: accessing a location that is now found in the cache
  - Note: locality only occurs when there is reuse, but reuse doesn't always result in locality
  - Steps
    1. Find data reuse
    2. Determine localized iteration space: set of inner loops where data is expected to fit within the cache
    3. Reuse ∩ Localized iteration space = Locality

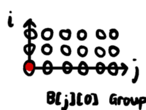
- Types of Data Reuse/Locality

```
for i=0 to 2
  for j=0 to 100
    A[i][j] = B[j][0] + B[j+1][0]
```

A[i][j] Spatial



B[j+1][0] Temporal



B[j][0] Group

- **Reuse Analysis**
  - Map n loop indices into d array indices to **map time into space**
    - $f(\vec{i}) = H\vec{i} + \vec{c}$

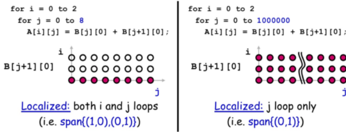
$$\begin{matrix} A[i][j] \\ B[j][0] \\ B[j+1][0] \end{matrix} = \begin{matrix} A\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}\begin{pmatrix}i\\j\end{pmatrix}+\begin{pmatrix}0\\0\end{pmatrix}\right) \\ B\left(\begin{pmatrix}0&1\\0&0\end{pmatrix}\begin{pmatrix}i\\j\end{pmatrix}+\begin{pmatrix}0\\0\end{pmatrix}\right) \\ C\left(\begin{pmatrix}0&1\\0&0\end{pmatrix}\begin{pmatrix}i\\j\end{pmatrix}+\begin{pmatrix}1\\0\end{pmatrix}\right) \end{matrix}$$

  - **Temporal reuse** occurs between iterations $\vec{i}_1$ and $\vec{i}_2$ when
    - $H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$, equivalently, $H(\vec{i}_1 - \vec{i}_2) = \vec{0}$, equivalently, reuse occurs along direction vector $\vec{r}$ when $H\vec{r} = \vec{0}$
    - So just compute the **nullspace of H**
  - **Spatial reuse** (for row major)
    - $H_s = H$ with last row replaced with 0
    - **Nullspace of H_s** gives vectors along which we access the same row

Lecture 3/18 cont.

- Reuse Analysis cont.
  - Group Analysis
    - Only consider uniformly generated sets where index expressions differ only by constant terms
    - Check if same cache line (constants could be too far apart)
    - Only leading reference suffers bulk of cache misses, so compiler focuses on that

- Localized Iteration Space



- Locality
  - Reuse vector space ∩ Localized Vector Space ⇒ Locality Vector Space

---

Lecture 3/23, 3/24, 3/25, 3/30

- Paper discussions, project meeting. Also future textbook readings seem very random, not making notes for those.

Lecture 3/31 Prefetching Arrays

- Memory latency
  - Reduce : locality optimizations as before
  - Tolerate : prefetching

- Prefetching
  - Overlap memory accesses with computation and other accesses
  - Types
    - Cache blocks : - limited to unit-stride accesses
    - Non-blocking loads : - limited ability to move back before use (run out of registers)
    - Hardware-controlled : - limited to constant strides, branch predictions + no instruction overhead
    - Software-controlled : - complexity, overhead + minimal hardware support needed, broader coverage
  - Concepts
    - Possible only if addresses can be determined ahead of time
    - Coverage factor = fraction of misses prefetched
    - Unnecessary if data already in cache
    - Effective if data in cache when later referenced
  - Two main concerns
    - Analysis : what to prefetch (max coverage factor, min unnecessary prefetch) ← misses are expensive but typical hit rate is >50%, avoid stressing memory
    - Scheduling : when to prefetch (max effectiveness, min overhead per prefetch)

Lecture 3/31 cont.

- **Prefetch Predicate** : prefetch when predicate is true

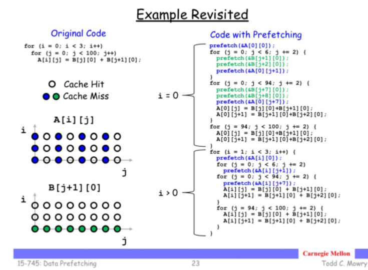
Locality Type	Miss Instance	Predicate
None	Every iteration	True
Temporal	First iteration	$i=0$
Spatial	Every ℓ iterations	$i \bmod \ell = 0$

$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix} \rightarrow j \bmod 2 = 0$

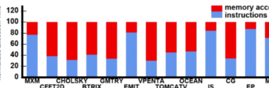
- **Loop splitting**
  - Decompose loop to isolate cache miss instances, cheaper than if statement

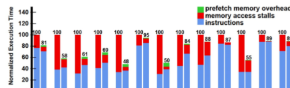
Locality Type	Miss Instance	Loop Transformation
None	Every iteration	None
Temporal	First iteration	Peel loop i
Spatial	Every ℓ iterations	Unroll loop by i

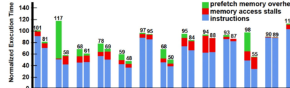
- **Software pipelining**
  - Iterations ahead = $\left\lceil \frac{\text{memory latency}}{\text{shortest path through loop body}} \right\rceil$
  - Loop → Prolog + Steady State + Epilog

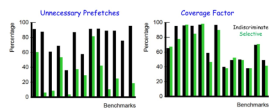


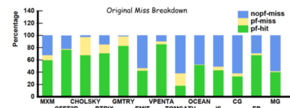
- Experimental results

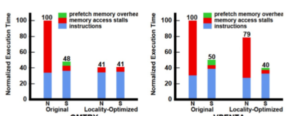




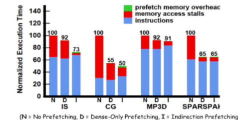








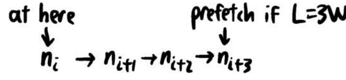
- **Prefetching indirection** A[index[i]]
  - Analysis : heuristic that assumes hit/miss (dense/sparse)
  - Scheduling : prefetch index ℓ iterations ahead
    - If 5 cycles, prefetch(&index[i+10]); prefetch(&A[index[i+5]]);



- Conclusion
  - Software prefetching effective
  - Hardware should focus on providing sufficient memory bandwidth

Lecture 4/1 Prefetching Pointer-Based Structures

- Goal: fully hide latency, compute at 1/W rate
  - L = loading a node
  - W = work
  - If L=3W, prefetch 3 nodes ahead
  - Without prefetching, rate = $\frac{1}{L+W}$
  - Prefetching 1 ahead, rate = $\frac{1}{L}$
  - Prefetching n ahead, rate is still $\frac{1}{L}$
  - Pointer chasing is limited to $\frac{1}{L}$, each iteration must still fetch pointer



- Pointer-chasing
  - Key: n_i needs $\&n_{i+d}$ without referencing d-1 intermediate nodes
  - Three proposals
    - Greedy: use existing pointers in n_i to approximate $\&n_{i+d}$
    - History-pointer: add new pointers to n_i to approximate $\&n_{i+d}$
    - Data linearization: compute $\&n_{i+d}$ directly from $\&n_i$

- Greedy
  - Not for linked lists, but e.g., tree
  - Prefetch all neighboring nodes, hope others visited later
  - Reasonably effective in practice, but little control over prefetching distance

- History-Pointer
  - First time adds history pointers
  - Subsequent traversals use the history pointers
  - Assumes past predicts future, trade space and time for prefetch distance

- Data-Linearization
  - If traversal order known, map nodes close in traversal to contiguous memory

- Summary, Experimental Results

Summary of Prefetching Algorithms

	Greedy	History-Pointer	Data-Linearization
Control over Prefetching Distance	little	more precise	more precise
Applicability to Recursive Data Structures	any RDS	revisited; changes only slowly	must have a major traversal order; changes only slowly
Overhead in Preparing Prefetch Addresses	none	space + time	none in practice
Ease of Implementation	relatively straightforward	more difficult	more difficulty

- Greedy prefetching is the most widely applicable algorithm
  - fully implemented in SUIF

15-745: Data Prefetching 49 Carnegie Mellon Todd C. Mowry

Performance of Compiler-Inserted Greedy Prefetching

O = Original
G = Compiler-Inserted Greedy Prefetching

- Eliminates much of the stall time in programs with large load stall penalties
  - half achieve speedups of 4% to 45%

15-745: Data Prefetching 53 Carnegie Mellon Todd C. Mowry

Coverage Factor

nopf_miss = original D-cache misses that are not prefetched
pf_miss = original D-cache misses that are prefetched but remain misses
pf_hit = original D-cache misses that are prefetched and then hit in the D-cache

- coverage factor = pf_hit + pf_miss
- 7 out of 10 have coverage factors > 60%
  - em3d, power, voronoi have many array or scalar load misses
- small pf_miss fractions → effective prefetch scheduling

15-745: Data Prefetching 54 Carnegie Mellon Todd C. Mowry

Unnecessary Prefetches

100 = all unnecessary dynamic pfs
99 = exclude all static pfs with hit rates > 99%

- % dynamic pfs that are unnecessary because the data is in the D-cache
- 4 have >80% unnecessary prefetches
- Could reduce overhead by eliminating static pfs that have high hit rates

15-745: Data Prefetching 55 Carnegie Mellon Todd C. Mowry

Reducing Overhead Through Memory Feedback

G = greedy prefetching
Fxx = greedy prefetching where static pfs with hit rate > xx% are eliminated

- Eliminating static pfs with hit rate >95% speeds them up by 1-8%
- However, eliminating useful prefetches can hurt performance
- Memory feedback can potentially improve performance

15-745: Data Prefetching 56 Carnegie Mellon Todd C. Mowry

Performance of History-Pointer Prefetching

O = original
G = greedy prefetching
H = history-pointer prefetching

- Applicable because a list structure does not change over time
- 40% speedup over greedy prefetching through:
  - better miss coverage (64% -> 100%)
  - fewer unnecessary prefetches (41% -> 29%)
- Improved accuracy outweighs increased overhead in this case

15-745: Data Prefetching 57 Carnegie Mellon Todd C. Mowry

Performance of Data-Linearization Prefetching

O = original
G = greedy prefetching
D = data-linearization prefetching

- Creation order equals major traversal order in treeadd & perimeter
  - hence data linearization is done without data restructuring
- 9% and 18% speedups over greedy prefetching through:
  - fewer unnecessary prefetches:
    - 94%->78% in perimeter, 87%->81% in treeadd
  - while maintaining good coverage factors:
    - 100%->80% in perimeter, 100%->93% in treeadd

15-745: Data Prefetching 58 Carnegie Mellon Todd C. Mowry

Lecture 4/6 Array-Dependence Analysis

- Four types of data dependence
  - Flow (true) dependence $S_i \, \delta^f \, S_j$ RAW
  - Anti dependence $S_i \, \delta^a \, S_j$ WAR
  - Output dependence $S_i \, \delta^o \, S_j$ WAW
  - Input dependence $S_i \, \delta^I \, S_j$ RAR

- **Value-oriented** : preserve values is enough ← { A=1, A=1, B=A+2 } ok
- **Location-oriented** : our focus, oblivious to values

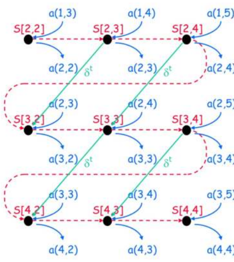
- **Dependence distance**
  - Same loop iteration, loop-independent dependence, $S_i \, \delta_0 \, S_j$ or $S_i \, \delta_= \, S_j$, direction is =
  - Flows between loop iterations, loop-carried dependence, $S_i \, \delta_c \, S_j$ or $S_i \, \delta_1 \, S_j$, direction is < aka positive

Example 4

```
do i = 2, 4
  do j = 2, 4
S:    a(i,j) = a(i-1,j+1)
  end do
end do
```

- An instance of S precedes another instance of S and S produces data that S consumes.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is (1,-1).

$S\delta^t_{(<,>)}S$ or $S\delta^t_{(1,-1)}S$



Problem Formulation

- Dependence testing is equivalent to an integer linear programming (ILP) problem of 2d variables & m+d constraint!
- An algorithm that determines if there exits two iteration vectors $\vec{k}$ and $\vec{j}$ that satisfies these constraints is called a dependence tester.
- The dependence distance vector is given by $\vec{j} - \vec{k}$.
- The dependence direction vector is give by sign($\vec{j} - \vec{k}$).
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be exact. Otherwise it is in-exact.
- A dependence test must be conservative; if the existence of dependence cannot be ascertained, dependence must be assumed.

- Went over Lamport test, GCD test

Loop Parallelization

- A dependence is said to be carried by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is loop-independent.

```
          do i = 2, n-1
            do j = 2, m-1
δt(=,=)       a(i, j) = ...
              ...     = a(i, j)

δt(=,<)       b(i, j) = ...
              ...     = b(i, j-1)

δt(<,=)       c(i, j) = ...
              ...     = c(i-1, j)
            end do
          end do
```

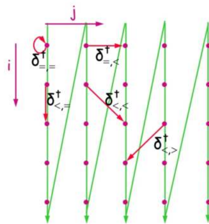
- Outermost loop with a non "=" direction carries dependence!

Loop Parallelization

The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop!

Loop Interchange

```
do i = 1,n
  do j = 1,n
    ... a(i,j) ...
  end do
end do
```




```
do j = 1,n
  do i = 1,n
    ... a(i,j) ...
  end do
end do
```

- When is loop interchange legal? when the "interchanged" dependences remain lexicographically positive!

Summary

- **Array data dependence testers:**
  - Use a cascaded approach, performing cheaper tests first
  - Summarize dependences with respect to surrounding loops:
    - may produce distances (1,-1) or directions (<,>)
- **When is it safe to run a loop in parallel?**
  - When that loop does not have a loop-carried dependence
    - outermost loop with a direction other than "="
    - (combine together for all array references in loop nest)
- **When is it legal to interchange or block/tile loops?**
  - When all dependences remain lexicographically positive
    - outermost direction other than "=" must be "<" (positive)
    - otherwise, sink would occur before the source

Lecture 4/7 Thread-Level Speculation

- Sounds like OCC for hardware, check then retry or commit
- Dataflow for scheduling
  - Stack : find instructions to compute forwarded value
  - Earliest : earliest node to compute forwarded value
- See slides or paper for details