**Review: Dominators**

- \( X \text{ dom } Y \) iff every execution path from the entry to \( Y \) goes through \( X \)
- Solved by simple fwd dataflow:
  - meet: intersection (only dominated by \( X \) if all predecessors are dominated by \( X \))
  - transfer: add myself

**More Review: Natural Loops**

- Defined by a backedge \( Y \to X \) where \( X \text{ dom } Y \)
- Finds (only) single-entry loops.
- Body: \( X \) plus those blocks that can reach \( Y \) without going through \( X \).
- Will find nested loop structure

**Not all cycles are natural loops**

- "irreducible", "improper", not "well-structured"...
- a multi-entry loop
- a CFG is "well-structured" iff its edge set can be partitioned into forward edges that form a DAG, and backedges according to our natural loop definition (the head dominates the tail)
Ok, let's find all the cycles

- Actually, usually want to find strongly connected components (SCC)

- SCC: every node in the SCC can reach every other node in that SCC by some directed path

- Can SCCs be nested?

- SCCs important in many areas - e.g. for cyclic scheduling, you want to find the SCCs in the DFG

- Singletons - not part of a cycle, their own SCC

---

Finding SCCs: Tarjan’s Algorithm

```c
visit(v) {
    N[v] = c; /* Mark v visited by assigning it a visit number. */
    L[v] = c; /* Low-link initially equal to visit number. */
    c++;
    push v onto the stack;
    for each w in OUT(v) {
        if N[w] == UNDEFINED { /* N[w] == UNDEFINED means w is unvisited. */
            visit(w); L[v] = min(L[v], L[w]); /* Low-link number can propagate upward. */
        } else if w is on the stack {
            L[v] = min(L[v], N[w]);
        }
    } /* Check if SC component found. */
    if L[v] == N[v] {
        pop vertices off stack down to v; /* These make up an SC component. */
    }
}
```

---

every node belongs to exactly one SCC
Finding SCCs: Tarjan's Algorithm

```
main_program {
  c := 0;    /* c is the counter for visit numbers. */
  for each vertex, v, in the graph,
    N[v] = UNDEFINED  /* Mark v "unvisited". */
    visit(v0);  /* v0 is the starting vertex. */
}
```
Finding SCCs: Tarjan's Algorithm

visiting: c
stack: c b a
e sees visited successor c; $L[c] = \min(L[e], N[c])$

Finding SCCs: Tarjan's Algorithm

visiting: e
stack: e c b a
e sees visited successor b; $L[e] = \min(L[e], N[b])$

Finding SCCs: Tarjan's Algorithm

visiting: f
stack: f e c b a

Finding SCCs: Tarjan's Algorithm

visiting: e
stack: e c b a

Finding SCCs: Tarjan's Algorithm

visiting: f
stack: f e c b a
Finding SCCs: Tarjan's Algorithm

visiting: g

stack: g f e c b a

Finding SCCs: Tarjan's Algorithm

visiting: h

stack: h g f e c b a

Finding SCCs: Tarjan's Algorithm

visiting: i

stack: g f e c b a

Finding SCCs: Tarjan's Algorithm

i sees visited successor h; \( L[i] = \min(L[i], N[h]) \)
Finding SCCs: Tarjan's Algorithm

visiting: j

stack: a b c d e j h g f e c b a

Finding SCCs: Tarjan's Algorithm

leaving: j

stack: a b c d e j h g f e c b a

Finding SCCs: Tarjan's Algorithm

leaving: i

stack: a b c d e j h g f e c b a

Finding SCCs: Tarjan's Algorithm

leaving: h

stack: a b c d e j h g f e c b a

j sees N=L; pops stack to create SCC {j}

i sees N != L; does nothing

h sees N = L; pops stack to form SCC {i,h}
Finding SCCs: Tarjan's Algorithm

leaving: g

stack: a b c

g sees N = L; pops stack to form SCC \{g\}

leaving: f

stack: a b c

f sees N = L; pops stack to form SCC \{f\}

leaving: e

stack: a b c

e sees N != L; does nothing

c updates L[c] = \min(L[c], L[e])

leaving: c

stack: a b c

c updates L[c] = \min(L[c], L[e])

sees L != N; does nothing
Finding SCCs: Tarjan's Algorithm

visiting: d

stack: d e c b a

stack contents: interesting...

Finding SCCs: Tarjan's Algorithm

leaving: d

d sees N != L; does nothing

stack: d e c b a

Finding SCCs: Tarjan's Algorithm

leaving: b

b updates L[b]=min(L[b],L[d]);
sees N = L; pops stack down to b
SCC = {d,e,c,b}
But what if you want more detail?

- Structural and Interval analysis: recognize and categorize both cyclic and acyclic control patterns.
- Form nested regions, each of which has a pattern type
- While forming regions, collapse each region to a supernode
- Hopefully irreducible regions can be quarantined to a small area
But what if you want more detail?

• Structural and Interval analysis: recognize and categorize both cyclic and acyclic control patterns.
• Form nested regions, each of which has a pattern type
• While forming regions, collapse each region to a supernode
• Hopefully irreducible regions can be quarantined to small area
But what if you want more detail?

- Structural and Interval analysis: recognize and categorize both cyclic and acyclic control patterns.
- Form nested regions, each of which has a pattern type
- While forming regions, collapse each region to a supernode
- Hopefully irreducible regions can be quarantined to small area
But what if you want more detail?

- Structural / Interval analysis: recognize and categorize both cyclic and acyclic control patterns.
- Form nested regions, each of which has a pattern type
- While forming regions, collapse each region to a supernode
- Hopefully irreducible regions can be quarantined to small area

Some Patterns

- if-then
- if-then-else
- block
- do-while
- while
- improper

Some Patterns

- general proper acyclic
- ...can’t have a template for every example, so have some general categories to catch the misfits...

Some Patterns

- general proper acyclic
- general improper cyclic
Control tree
- Can build a tree of the nested regions:
  - each node is a region
  - leaves are basic blocks
  - the root is the entire procedure
  - a region’s parent is the immediately enclosing region

T1-T2 Reduction
- Oldest and simplest
- Can reduce all well-structured graphs!

T1-T2 Example
- Hierarchy can seem strange....

T1-T2 Reduction
- But...cannot reduce irreducible graphs!
  "end up w/ "limit flow graph"
T1-T2 Example

- Hierarchy can seem strange....

(out edges from new region get merged – not shown)

T1-T2 Example

- Hierarchy can seem strange....

T1-T2 Example

- Hierarchy can seem strange....

T1-T2 Example

- Hierarchy can seem strange....
But why????

- Makes IR -> source conversion prettier....

But why...really????

- An alternate approach to dataflow analysis
  - before, we iterated on basic blocks
  - now, each time we form a region ->
    form a composite transfer function that
    *summarizes the effect of that region*

- Simple example:

\[
\begin{align*}
\text{fA}(x) &= \text{GenA} \cup (x - \text{KillA}) \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{fA}(x) - \text{KillB}) \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{GenA} \cup (x - \text{KillA})) - \text{KillB} \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{GenA} - \text{KillB}) \cup (x - (\text{KillA} \cup \text{KillB}))
\end{align*}
\]

Dataflow Analysis on the Control Tree

- After all regions are formed -
  when there is just one region for the whole proc -
  when you’ve reached the root of the control tree -
  you get one transfer function for the whole proc

- But what good is it to have dataflow info at the exit node?

- The rest of the story: you also build functions for
distributing the results back down the control tree to
each region, eventually to the leaves (basic blocks)

Details...

- How to calculate fB•fA?
- Well, we have already done this when computing the
  transfer function of a block that is a sequence of
  instructions...but to spell it out:

\[
\begin{align*}
\text{fA}(x) &= \text{GenA} \cup (x - \text{KillA}) \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{fA}(x) - \text{KillB}) \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{GenA} \cup (x - \text{KillA})) - \text{KillB} \\
\text{fB}(\text{fA}(x)) &= \text{GenB} \cup (\text{GenA} - \text{KillB}) \cup (x - (\text{KillA} \cup \text{KillB}))
\end{align*}
\]
More Sample Calculations

\[ f_R(x) = f_B(f_A(x)) \land f_A(x) \]
\[ = [(f_B \land f_A) \land f_A](x) \]
\[ = [(f_B \land I) \land f_A](x) \]

\( ^\land \) is the meet operator

• gets just slightly more complicated for flow-sensitive transfer functions where \( f_A_{\text{then}} \) is different than \( f_A_{\text{else}} \)

• distribution calculation (coming down the control tree) is obvious

Review

- Structural, Interval, or T1-T2: find nested regions and build the control tree
- Summarize transfer function for each region as you go up the control tree
- Evaluate
- Distribute results going back down the control tree
- Analogies:
  - solving system of equations by elimination
  - parallel prefix

But still ... why???

- Is this better than an iterative data flow solution?
  - Well, can be useful with incremental changes: could confine re-analysis to a small subtree of the control tree
  - Might be better than iterative for deeply nested graphs (if loop closures can be computed efficiently)
  - Historically, at the time this approach was developed, it was not recognized that iterative dataflow can be solved quickly if you visit the basic blocks in the correct order (fwd or bkw topological)

- But...
  - doesn’t handle irreducible areas well
  - backward dataflow problems - difficult!
    - iterative dataflow symmetric; dom/postdom symmetric;
    BUT, many CFGs are not reducible when reversed... why?
Another use for profiling: loop count

- A large class of loop optimizations improve the time per iteration but add a fixed overhead
- Characteristic break-even point

```
if (N > breakeven)
    [vector_loop];
else
    [non-vector_loop];
```

Another use for profiling: loop count

- Obvious approach: if average loop count (from profiling) is less than the break-even point, then use the un-optimized version
- But what if loop count varies greatly? ...and the average is near the break-even point?
- From vectorization: compile two versions of the loop

```c
if (N > breakeven)
    [vector_loop];
else
    [non-vector_loop];
```

Another use for profiling: loop count

- But...how do you know beforehand if the loop count varies? No profiling we’ve described summarizes variance of loop counts.
- And if there is no variance, the added code for two loop versions is useless code expansion, and the loop count check at the loop entry is useless overhead.
- So you want 2 versions ONLY when there’s variance
- Possible approaches:
  - special record of loop counts
  - whole program path
  - simple predictors (works even with WHILE loops)
  - dynamic optimization
When is run-time check worth the overhead?

- See also: Calpa – in reading list
- Uses compile-time analysis to decide where it is beneficial to add dynamic (run-time) checks for run-time re-optimization

Big Profiling Issue: Robustness

- Can your profile-driven optimization hurt if the actual data set differs much from the training data set?
- How much?
- Are you hosed?
- Can you buy insurance?

Profile-based gcc optimization

- `-fprofile-arcs`
  - Instrument arcs during compilation to generate coverage data or for profile-directed block ordering. During execution the program records how many times each branch is executed and how many times it is taken. When the compiled program exits it saves this data to a file called `sourcename.da` for each source file.

- `-fbranch-probabilities`
  - After running a program compiled with `-fprofile-arcs` (see Options for Debugging Your Program or `gcc` Options for Debugging Your Program), you can compile it a second time using `-fbranch-probabilities`, to improve optimizations based on the number of times each branch was taken.

- `-fno-guess-branch-probability`
  - Do not guess branch probabilities using a randomized model.
  - Sometimes gcc will opt to use a randomized model to guess branch probabilities, when none are available.

...done.
Outline I

• motivating example, other motivation (test coverage)
• can we exploit "probably" rather than "always"?
• common case fast - what is the common case?
• gprof, node, edge - brief how-to
• big pic - branch prediction is just hw profiling..trace..

• profile usage for standard optimizations
  - tail dup, superblock - cost is code expansion
  - just xform, then use existing opts
  - extended by ammons - actual benefit from duplication?

• hyperblock formation heuristic
  - will add paths as long as doesn’t impact main path
• my case - loops - kernel - excluding - prune points

Outline II

• probability quiz ---aka lies, damn lies, statistics
• edge profiles alone cannot predict common path
• path profiling use: branch correlation
  - san diego vs. pittsburgh
  - also important for test coverage
• efficient path profiling
  - built on earlier work to improve edge profiling

• common situation - per iteration savings, fixed overhead
• runtime test - worth the overhead?
• similar situation - calpa
• Data profiling?
• more general issue - robustness in the face of different datasets - or even different phases in the same dataset