Compilers in Context

Compilers are one tool in a programming environment

- Compiler
- Interpreter
- Editor
- Debugger
- Profiler
- …

Generation and Use of Dynamic Profiles

Dynamic execution profiles can show

- # of times procedure, stmt, branch executed
- run-time in procedure, stmt, branch
- # of times variable referenced
- …

Profiles can be used to

- identify hot spots
- verify correctness
- learn about program, input

How do you get profile data?

Tradeoff:

- accuracy
- run-time cost & invasiveness
- implementation complexity

Option 1: Instrumentation

- Insert counters in program to record interesting events

Option 2: Sampling

- At interrupts, examine state of the program

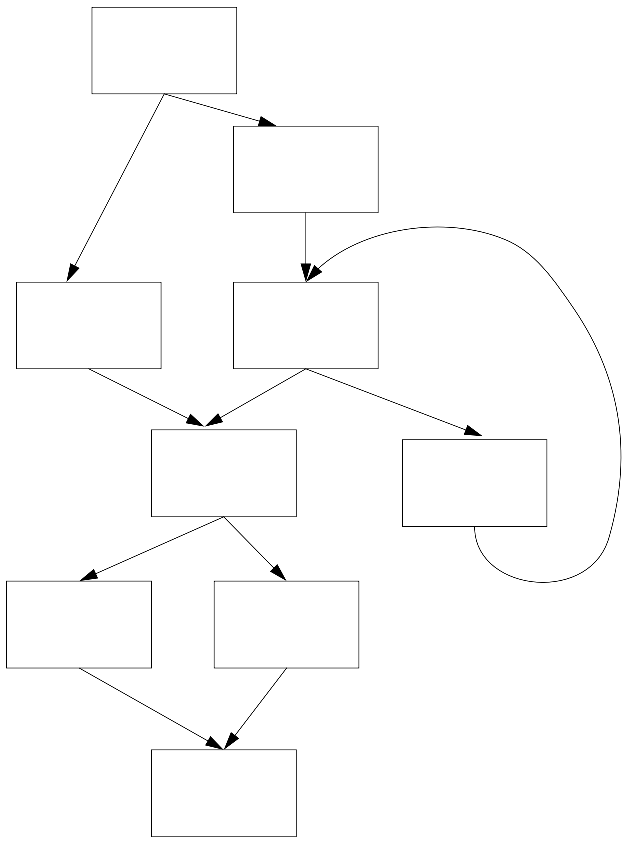
Option 3: Both

- instrument low frequency events for accuracy
- use sampling for frequent events, less accurate info

Counter-based Profiling

Suppose we wanted accurate dynamic instruction counts…

1. Insert counter before every instruction
   - Simple, slow
2. Only 1 counter per basic block
   - simple, faster,
   - `pixie`
3. Omit counters for simple merge basic blocks
   - calculate by summing predecessor
   - still need counters at loop heads
4. Omit counters at 1/2 of branch successors
   - can compute from sibling & predecessor count



Optimal placement of counters

[Ball & Larus 92]

Suppose we were given a measured/estimated execution frequency for all bb's and/or arcs between bb's.

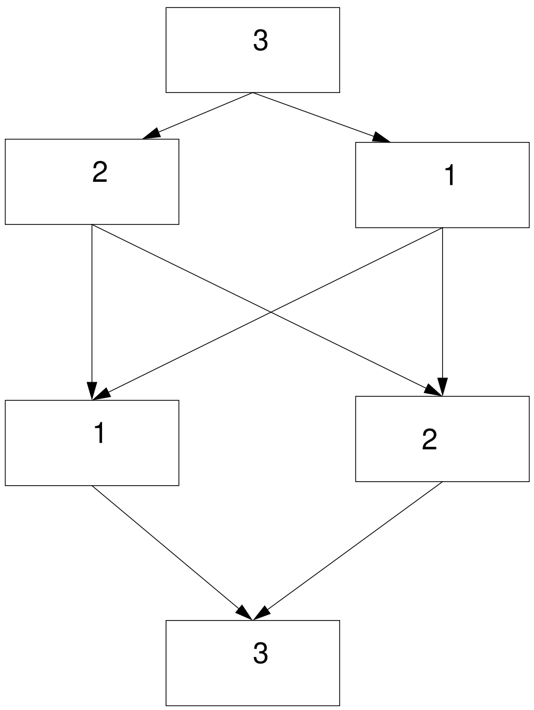
We want to get a complete, accurate trace of the program.

Where should counters be placed such that expected overhead (overhead == incrementing counters) is minimized?

Variations:

- given arc freq or bb freq?
- want arc count or bb count?

Can go from arcs to bb's, cannot always go from bb's to arcs.

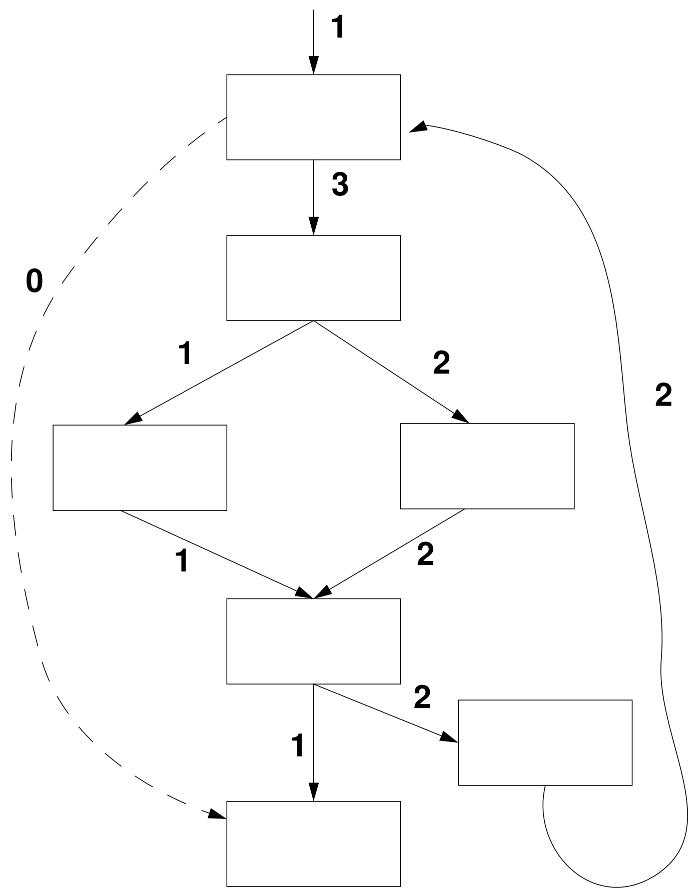


Algorithm

1. Find highest-weight spanning tree of CFG
2. Instrument all other arcs

Results:

- optimal (wrt weighting) placement of counters to solve edges from edges problem.
- applied to SPEC benchmarks
  - 2-4 times lower overhead than `pixie`
  - (4-50% overhead total)



Displaying profile data

Problem: Get data into useful format

Option 1: flat profile (e.g. `prof`)

- table of results, sorted by weight
- each procedure treated separately
- counts per instruction, basic block, procedure

Option 2: hierarchical profile (e.g. `gprof`)

- include time spent in callees in caller's time
- attempt to reflect cost of entire abstraction

Calculating a hierarchical profile

[Graham, Kessler & McKusick 82]

flat profile + dynamic call graph ⇒ hierarchical profile

Get flat profile by usual means (gprof uses sampling)
Get dynamic call graph by instrumenting call-sites

- When `r` calls `e`, increment entry for `r`→`e`

Building hierarchical profile

Construct weighted call graph

- use measured $r \rightarrow e$ counts
- augment with 0-count arcs by examining executable

For each routine r, calculate total hierarchical time T_r as follows:

$$T_r = S_r + \Sigma_{r \rightarrow e}\, T_e * (C_e^r / C_e)$$

where

C_e^r = call count of $r \rightarrow e$

$C_e = \Sigma_r\, C_e^r$

Solving the recurrence

If call graph is a DAG, solve using reverse topological traversal

If call graph has cycles:

- collapse strongly connected components into single node
  - combine execution times
  - merge call arcs
- solve as a DAG
- results can be ugly
  - for example, user defined control structures can make the entire program look like a recursive cycle.

Weakness of this approach

Each caller is charged for a percentage of the callee's time based on the percentage of calls it made.

What if all calls are not equal?

An alternative:

- periodically (timer interrupts?) examine stack
- record stack dump (possibly compressed)
- derive hierarchical profile from this data

Cool Tools

Tool-building tools

Shade [Cmelik & Keppel 93]
- uses instruction level simulation

Atom [Srivastava & Eustace 94]
- object code modification

Both allow users to supply small pieces of code that leverage of off the provided infrastructure.

Compilers as consumers of profile data

Compilers could use profile data to guide optimization
- Impact C compiler -- inlining
- Vortex -- inlining, specialization
- Wall -- allocating globals to registers, “super optimizer”
- scheduling, …

Does this really work?
- Information must be useful
- Information must be stable

Stability of profiles

How well do profiles predict future behavior of program?

[Wall 91]:
Looked at several static estimation techniques and profiles

Predictions for:

- execution frequency of call, bb, proc, ref to global
- time in proc

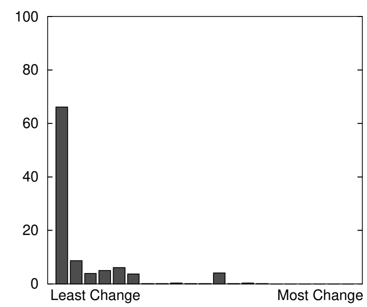
Results:

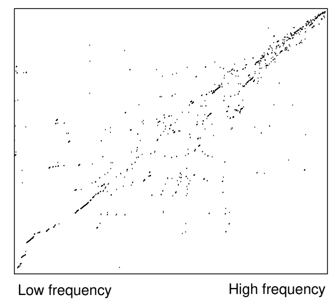
- static estimates awful (<25% “effectiveness”)
  - random was almost as good as some static estimates
  - effectiveness varied on property, percentile
- profiles ok (about 75% “effectiveness”)

[Garrett et al 93]

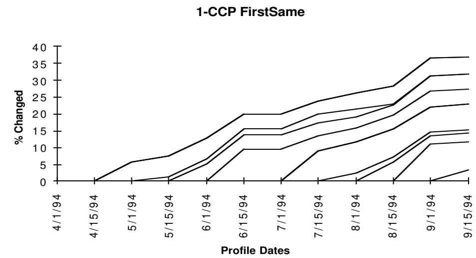
Looked at stability of class distributions used for receiver class prediction for OOL’s (Cecil).

- Cross input stability





- Cross version stability



Debugging Support

What do you expect the debugger to be able to do?

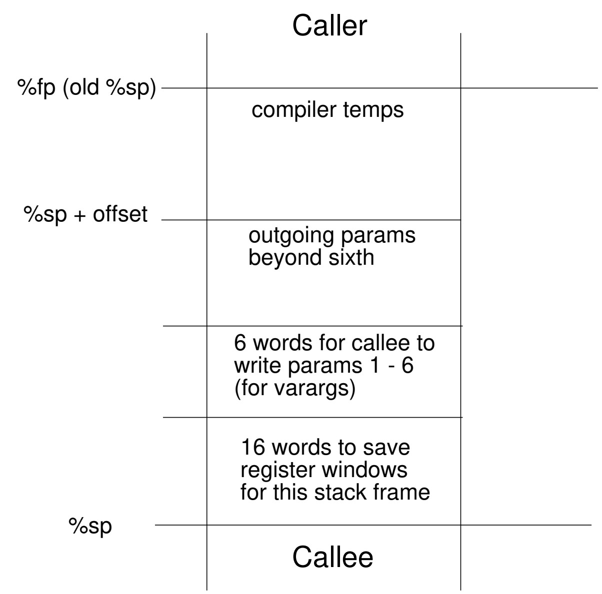
- display value of program variables
- display current stmt
- display call chain
- breakpoints, single step, tracing
- modify variables and continue and/or restart
- modify code and continue and/or restart

Want this at no cost when not used

Stack layout (simplified SPARC)

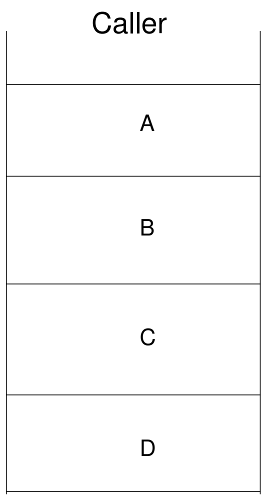
governed by conventions

register windows, no callee saved registers



What about callee saved registers?

Where are they stored?



1. each procedure stores bitmask of used callee saved registers in a known location
2. callee saved registers are saved on entry, restored on exit
3. debugger needs to know whether in entry/exit sequence

What does the debugger need to know?

Source/target mappings

- Code location:
  - mapping between source and machine stmts
- Stack location
  - mapping between virtual and physical stack frames
- Data location:
  - mapping between source variables & stack/registers

Mappings are bi-directional

The compiler is responsible for generating mappings

What can go wrong?

```
k = 15;
for(i=0; i != 9; i++) {
  for(j=i+1; j!=10; j++) {
    if(a[i] > a[j]) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
    }
  }
}
```

```
if(...) {
  x = 6;
  y = q;
} else {
  x = z;
}
```


Optimizer *vs.* debugger

Mappings straightforward for unoptimized code

Problem: Optimization makes life hard

Solutions:

- Optimizer wins, debug as much as possible [gcc, DOC]
- Debugger wins, optimize as much as possible [SELF, Cecil]

Why debug optimized code?

Code location problem

Debugger wants to:

- map current PC to source location
- single step
- set breakpoints
- handle traps/exceptions
- modify code and continue/restart

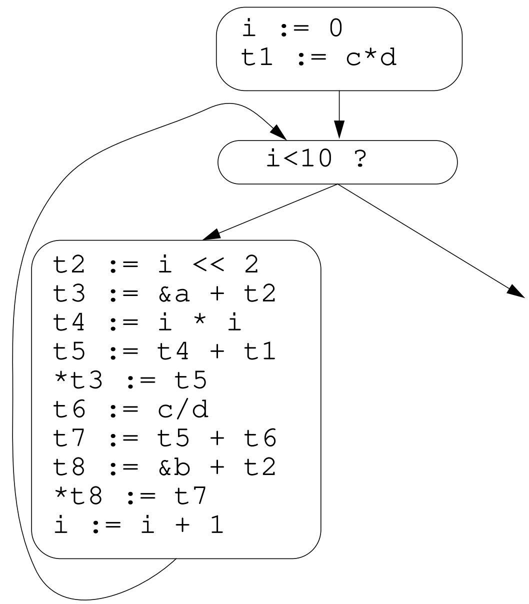
But, the optimizer can:

- change statement ordering
- intermingle statements
- change procedure boundaries

Preserve source-level view of stmt order?

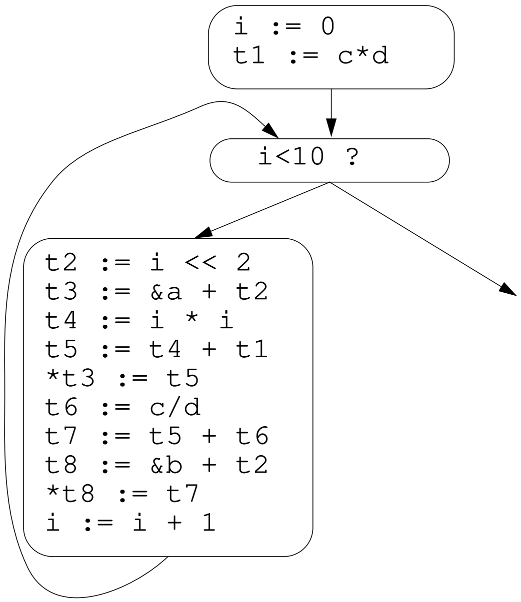
Solving code location

```
1:  for(i=0; i<10; i++) {
2:    a[i] = i*i + c*d;
3:    b[i] = a[i] + c/d;
    }
```



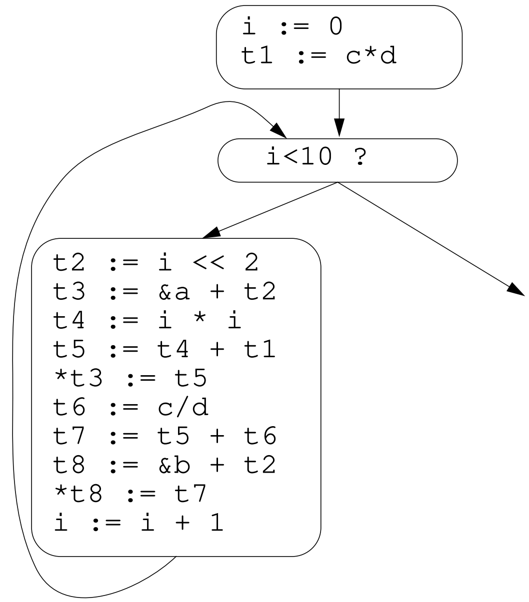
Tag every instruction

```
1:  for(i=0; i<10; i++) {
2:    a[i] = i*i + c*d;
3:    b[i] = a[i] + c/d;
    }
```



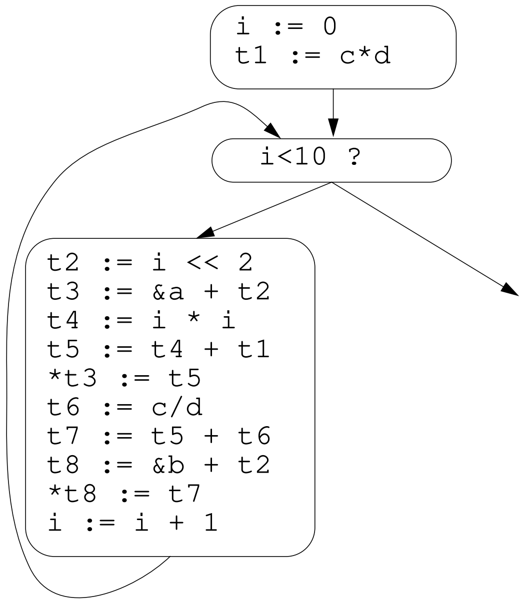
Tag ‘first’ instruction of each stmt

```
1:  for(i=0; i<10; i++) {
2:    a[i] = i*i + c*d;
3:    b[i] = a[i] + c/d;
    }
```



Tag ranges of instructions

```
1:  for(i=0; i<10; i++) {
2:    a[i] = i*i + c*d;
3:    b[i] = a[i] + c/d;
    }
```



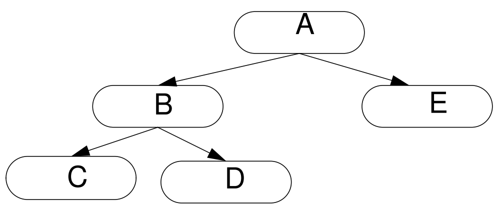
Interruptible points

- Cecil, SELF
- map function calls, loop heads to source locations
- freely schedule between interruptible points
- may reduce to tagging every instruction depending on definition of interruptible point

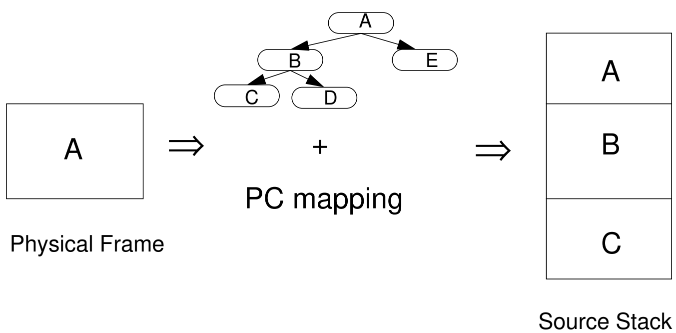
Stack location problem

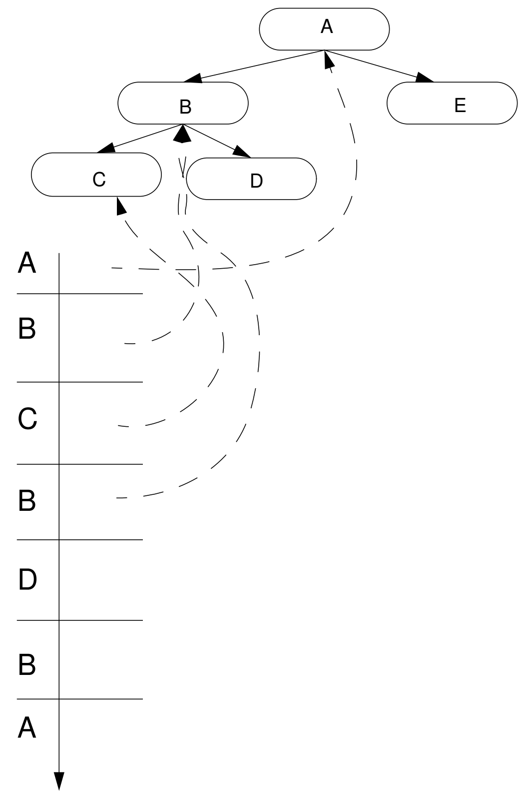
Mapping physical and virtual stack frames

- Inlining can merge virtual frames [Self]
- Tail call elimination can wipe out virtual frames



Suppose I set a breakpoint in C,





Code modification

Changed procedure, want to continue

1. Procedure not on stack
   - dynamic linking, runtime code generation, interpretation
2. Procedure on stack?
   - continue with old procedure
   - cut back stack
   - modify stack

Data value problem

Debugger wants to

- Look up the meaning of an identifier
- Display the value of a source variable
- Modify the value of a source variable and continue/restart

But, the optimizer can:

- remove variables
- remove updates to variables
- make variables constant
- make variables share a location
- put variables in different locations
- put variables in mulitple locations

Can source variables always be displayed/modified?

Solving data value

- Restrict variables to a single location
- Map PC ranges to location
  - need live ranges, just like register allocation

Possible locations:

- on stack (offset of sp)
- in register (reg number)
- constant (constant value)
- computable from other variables (expr)
- unavailable

Modifying data value

Use mapping to find location(s) of variable
modify & continue

But,

- What if location is shared by multiple variables?
- What if variable was a constant?
- What if variable was used in CSE downstream?

SELF used dynamic compilation to produce unoptimized code that was run after data modification.

Deoptimization

1. Save contents of physical stack frame & pop it
2. Determine virtual activation records
3. For each virtual activation record, compile a completely unoptimized routine, create a physical stack frame
4. patch up PC & run-time system pointers in activation frame
5. init each virtual activation record with values from original saved physical frame

Start running again...

What if stack frame to be deoptimized is not on top?

Can't just bash stack, do it lazily…

Change return address for callee of target frame