7.6 Practical issues

7.6.1 Instrumentation

Edge profile $\rightarrow$ Block profile

Set of counters for edges: $E_{count}$
Set of counters for blocks: $B_{count}$

Why bother?

Cost of instrumentation differs: edges/blocks require different steps

$B' \in B_{count}$: insert "incr count"
today: \text{cost of counting edges} \leq \text{cost of counting blocks}

\textit{(Kirchhoff's Law)}

\begin{align*}
\text{Freq (block } B\text{)} &= \sum \text{Freq (E)} \\
&= \sum \text{Freq (E)} \\
&= \sum \text{Freq (E)}
\end{align*}

\text{E incoming edge of } B

\text{E outgoing edge of } B

\text{subject to change…}
CFG with set of blocks $B$
edges $E$

Spanning tree: given graph $(B, E)$
such that in $(B, E)$
every pair $B', B''$ in $B$ is connected
by a unique path.

connects all blocks, no cycles

(direction of edge does not matter when building the spanning tree)

Weight on edge: maximum spanning tree
S.t. such that weights on edges in $E$
are maximal

(efficient algorithms exist)
void foo() { 
while (P) do { 
if (Q) { 
    A 
} else { 
    B 
} 
if (R) { 
    exit() 
} 
} 
C 
} 

To ensure that K's Law is valid:
ADD edge Exit → Start
max spanning tree

all pairs of nodes are reachable

(C, Exit):

C → P

Exit → P

C → P → Exit

Idea: use the maximum spanning tree to find edges to instrument avoid max SP tree edges
max. sp. tree: block

0: option 1
   cost
   0 + 3 + 1 + 1 = 5

0: option 2
   cost
   0 + 1 + 2 + 1 = 4

(exit heuristics -- but for many CFGs: optimal solution exists)
Code generation for instrumentation

2 Approaches

1) Modify Compiler
   + instrumentation code optimized (register allocation)
   - adds complexity
   multi-language programs/libraries a pain

2) Binary rewriting tool
   + works for all object files
   code placement easy
   - opt like reg curly difficult
   must be able to build CFG
7.6.2 Path profiles

Path: sequence of blocks

Basic idea: for each block put a marker ("stamp") into a file

Witness

Process — all execution of the program — this file

Witness file

Same thoughts (as discussed in 7.6.1)

Apply here: edge witness collection — block witness file
Reconstruction of block sequence is easy

Given a witness file WF

```python
block = START node (first block of prog.)
print (block)

while (! EOC (WF))
    if ((block == EXIT) || (block has single successor s_block))
        block = s_block
    else
        block = read (WF)
print (block)
```

⑨ ③
Witness W1

Witness W1

Witness W2

Mark/record method/fct calls...
7.7 Code positioning

Cache is fast

Memory is slow

\( \rightarrow \) small organized as a collection of storage cells

- have same property
  - either held data "valid"
  - or not "invalid"

Cache lines

Cache lines are filled together

Access to some location \( L \), until \( L \) mapped to cache line \( C_L \)

All storage cells in \( C_L \) become valid.
Idea: make sure that all items fetched from memory and put into cache line Cl are useful for program execution (paid for transfer!)

(Item is used before it is flushed from the cache)

Compiler should try to keep items that are used together in neighboring memory locations - access to one item brings all of them into the cache.

- Instruction space seems to model the data space - for now: look at instructions
7.7.1 Code positioning for functions

group methods that (often) executed together
so that the cache management brings in code
from methods that are executed soon.

position code in the instruction segment

need: information about methods called
together

call graph with frequency information
about actual calls.
Functions $A, B$

$A \rightarrow B$
calls

$\alpha$

$B \rightarrow A$
calls

Simple call graph (example)

weight: number of calls (how often: $A \rightarrow B$)

(smaller weights for ease of presentation...)
weighted call graph provides guidance on how to put code into for functions into memory. Simple scheme works (surprisingly) well.

Cluster "freq-exec" methods.

Take WCcall graph:

Replace pair of nodes \((P, Q)\) in call graph by combined node \(C_{PQ}\) if:

1) edge between \(P \rightarrow Q\)

2) and \(W_{pq}\) weighted

\(W_{pq} > Wxy\) if \(x\in Y\) call graph
Update call graph to reflect merging of nodes
(Two edges with equal weight... toss a coin)

Repeat until one node remains

Question: next is AD CF.
ADCF ADFC DA CF DAFC... what order??