Dynamic Dependence Analysis

• CFGs relevant for dynamic analysis: constructed at instrumentation time
  – For simplicity of presentation, we will discuss
    • Intraprocedural: one procedure/method
    • Nodes are individual three-address instructions rather than basic blocks

• Goal 1: tracing (already discussed)
• Goal 2: dynamic control dependences
• Goal 3: dynamic data dependences
Tracing and Dependences

• Each three-address instruction in the code is given an integer ID at instrumentation time
• The simplest possible trace is a sequence of trace events $te_i$: $\text{trace} = (te_0, te_1, ..., te_n)$
  – Each event contains an instruction ID
• Dependence: pair $(te_i, te_k)$ with $i<k$ such that the first event must happen before the second one
  – E.g., $te_i$ computes a value and writes it to memory; then $te_k$ reads this value from the same location in memory (data dependence)
Dynamic Dependence Analysis

• **Online**: as the instructions get executed, their dependences are discovered on the fly
  – Possible output: trace annotated with dependence info: each trace event has a list of prior events on which it is dependent
  – Another possibility: while the program is running, the on-the-fly dependences are used for correctness checking, computing various metrics, etc.

• **Offline**: just output the trace; after the run, the trace is analyzed for dependences
  – Need more info in the trace: e.g., if an instruction instance reads/writes a memory location, the memory address is recorded in the trace event
Dominance

- Detour into (mild) graph theory for static analysis
- A CFG node $d$ dominates another node $n$ if every path from ENTRY to $n$ goes through $d$
  - Implicit assumption: every node is reachable from ENTRY (i.e., there is no dead code)
- Many uses of this info
  - E.g., to perform analysis of loops in a CFG
    - Back edge: a CFG edge $(n,h)$ where $h$ dominates $n$
    - Natural loop for $(n,h)$: the set of all nodes $m$ that can reach node $n$ without going through node $h$ (trivially, includes $h$)
      - $h$ dominates all such nodes $m$
      - $h$ is the header of the natural loop
Post-Dominance

• A CFG node $d$ post-dominates another node $n$ if every path from $n$ to EXIT goes through $d$
  – Implicit assumption: EXIT is reachable from every node
  – A relation $pdom \subseteq \text{Nodes} \times \text{Nodes}$: $d \ pdom \ n$
  – The relation is trivially reflexive: $d \ pdom \ d$

• Post-dominance on a CFG is equivalent to dominance on the reverse CFG (all edges reversed)
Control Dependence: Informally

• A node \( n \) is control dependent on a node \( c \) if
  – There exists an edge \( e_1 \) coming out of \( c \) that definitely causes \( n \) to execute
  – There exists some edge \( e_2 \) coming out of \( c \) that is the start of some path that avoids the execution of \( n \)

• The decision made at \( c \) affects whether \( n \) gets executed: if \( e_1 \) is followed, \( n \) definitely is executed; if \( e_2 \) is followed, there is the possibility that \( n \) is not executed at all
  – Thus, \( n \) is control dependent on \( c \) – whether \( n \) gets executed depends on what \( c \) does
Control Dependence: Formally

• (part 1) $n$ is control dependent on $c$ (where $n \neq c$) if
  – $n$ does not post-dominate $c$
  – there exists a path from $c$ to $n$ such that $n$ post-dominates every node on the path except $c$

• (part 2) $n$ is control dependent on $n$ if
  – there exists a path from $n$ to $n$ (with at least one edge) such that $n$ post-dominates every node on the path
    • this implies that $n$ has two outgoing edges
    • this case applies to the header of a loop
Consider all branch nodes \( c \): 1, 4, 7, 8, 10

ENTRY does not post-dominate any other \( n \)
1 \( pdom \) ENTRY, 1, 9
2 does not post-dominate any other \( n \)
3 \( pdom \) ENTRY, 1, 2, 3, 9
4 \( pdom \) ENTRY, 1, 2, 3, 4, 9
5 does not post-dominate any other \( n \)
6 does not post-dominate any other \( n \)
7 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 9
8 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9
9 does not post-dominate any other \( n \)
10 \( pdom \) ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

EXIT \( pdom \) \( n \) for any \( n \)

2 is control dependent on 1
3, 4, 5, 6 are control dependent on 4
4, 7 are control dependent on 7
9, 1, 3, 4, 7, 8 are control dependent on 8
7, 8, 10 are control dependent on 10

Note: a node may be control dependent on several other nodes (e.g., node 3)
Dynamic Control Dependences

• Static control dependences are computed at instrumentation time

• Dynamic control dependence \((te_i, te_k)\) for \(i<k\)
  – Event \(te_i\) is an instance of CFG node \(c\)
  – Event \(te_k\) is an instance of CFG node \(n\)
  – Node \(n\) is statically control dependent on \(c\)
  – There does not exist an event \(te_j\) (for \(i<j<k\)) such that \(n\) is statically control dependent on the CFG node corresponding to \(te_j\)

• For any \(te_k\) there is a unique \(te_i\) with this property
  – Or, no such \(te_i\) exists
Online Detection of Control Dependences

- Goal: whenever we write an event $te_k$ to the trace, also write the control dep $(te_i, te_k)$ if it exists
- Maintain a global timestamp $TS$: the number of events produced up to this point
  - Initialized/incremented as necessary
- For each CFG node $c$ that is a branch, maintain extra info $last(c)$: the value of $TS$ recorded when the last instance of $c$ was executed
  - E.g., map integer instruction ID $\rightarrow$ integer timestamp
- When $te_k$ occurs: if the corresp. CFG node is $n$, look at all $c$ on which $n$ is statically control dependent, and pick the one with the largest value of $last(c)$
  - This largest timestamp is the $i$ for $te_i$
Static Data Dependence Analysis

• Goal: identify all connections between variable definitions ("write") and variable uses ("read")
  – \( x = y + z \) has a definition of \( x \) and uses of \( y \) and \( z \)

• A definition \( d \) reaches a use \( u \) if there exists a CFG path that (1) starts at \( d \), (2) ends at \( u \), and (3) does not contain a re-definition (i.e., \( d \) is not "killed")
  – Reaching definitions: standard compile-time analysis
  – Def-use pairs represent static data dependences

• Static analysis is good for scalar variables, but bad for arrays and pointers
  – E.g., \( a[t1] = \ldots \) and \( \ldots = a[t2] \), or \( *p = \ldots \) and \( \ldots = *q \)
Dynamic Data Dependence Analysis

• We cannot simply do what we did for control dep
  – Cannot just maintain timestamp $\text{last}(n)$ for each CFG node $n$, and look at all static data dependences

• Solution: for each memory location $m$ that could be read or written, maintain $\text{last}(m)$: the value of $\text{TS}$ recorded the last time $m$ was written
  – Implementation: shadow memory

• Whenever an event $te_k$ occurs: if this event reads $m$, the value of $\text{last}(m)$ is the value of $i$ for a dynamic data dependence $(te_i, te_k)$

• Many possible optimizations to reduce cost