Efficient On-the-fly Analysis of Program Behavior and Static Cache Simulation

Frank Mueller, David Whalley

Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

e-mail:
mueller@cs.fsu.edu

WWW:
http://www.cs.fsu.edu/~mueller

Objective

- provide faster cache performance evaluation
  - determine number of hits and misses of a program execution
  - used to evaluate new cache designs
  - used to analyze new optimization techniques
- predict the caching behavior (for real-time systems)

Methods in Contrast

- Goal: faster cache performance evaluation
- traditional approach: inline tracing
  - instrument program on complement of min. spanning tree
  - generate trace addresses
  - simulate caches based on trace
- our approach: **on-the-fly analysis**
  - analyze program statically (static cache simulation)
  - instrument program on "unique paths"
  - do NOT generate trace addresses
  - simulate remaining cache behavior within program execution

Small Set of Measurement Points

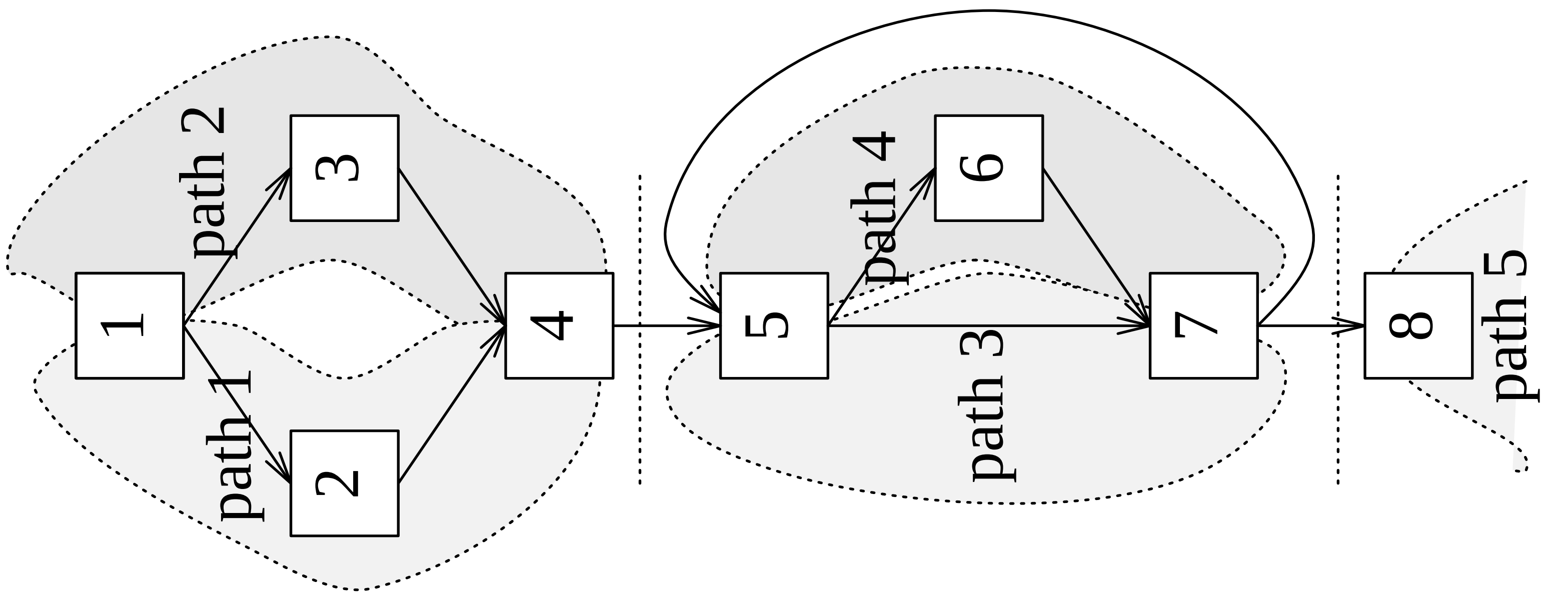
- covers all events and preserves their order during execution
- applicable for any on-the-fly analysis of program behavior
- cannot use min. spanning tree if order of events critical
- need new method to find a small set of measure points
- partition control-flow graph into *unique paths:*
  - unique transition for each path to place instrumentation code
  - path contains sequence of basic blocks in control flow
  - blocks in path not necessarily consecutive code
  - set of blocks determines portion of program for static analysis

Definition of Unique Path Partitioning (UPPA)

1. all vertices covered by paths
2. edges are either in paths or connect paths
3. each path has unique edge or vertex
4. paths overlap only in initial or final subpaths
5. paths are chained properly
6. calls terminate paths (operational)
7. paths do not cross loop boundaries (operational)

Properties of UPPAs

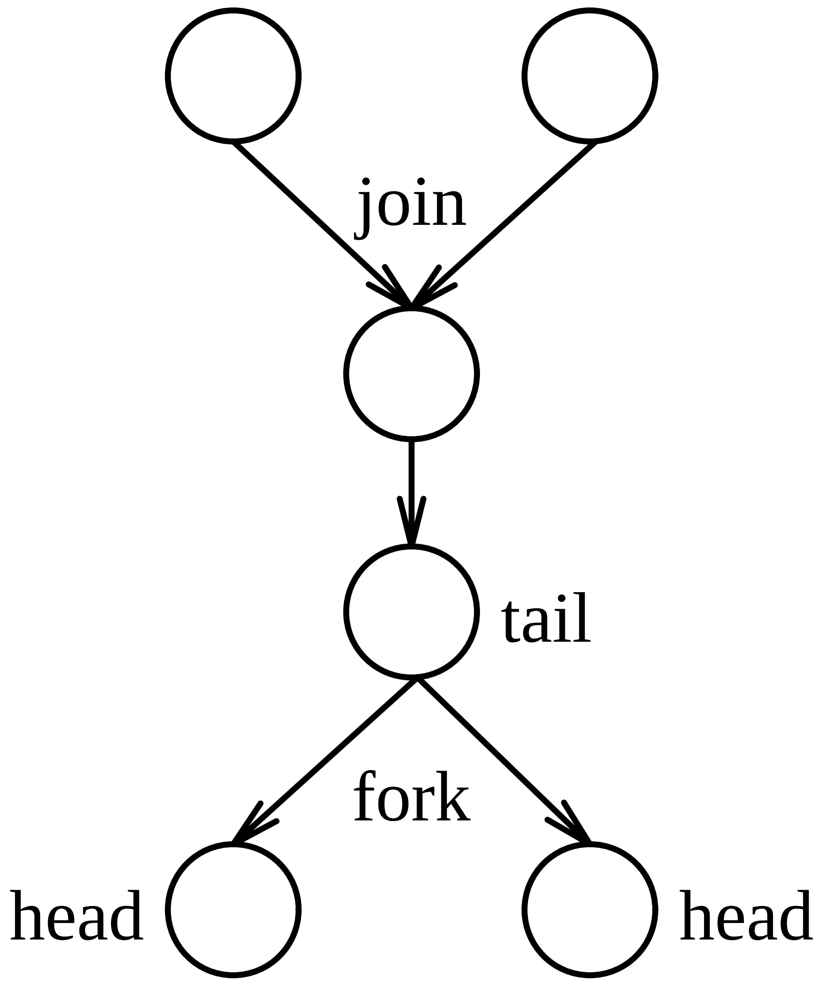
- basic block partitioning is a UPPA
- Let $|UPPA|$ denote number of paths in partitioning
- ordering: $UPPA_a < UPPA_b := |UPPA_a| < |UPPA_b|$
- goal: find minimal UPPA for a given control-flow graph


1
2
3
4
5
6
7
8
path 1
path 2
path 3
path 4
path 5

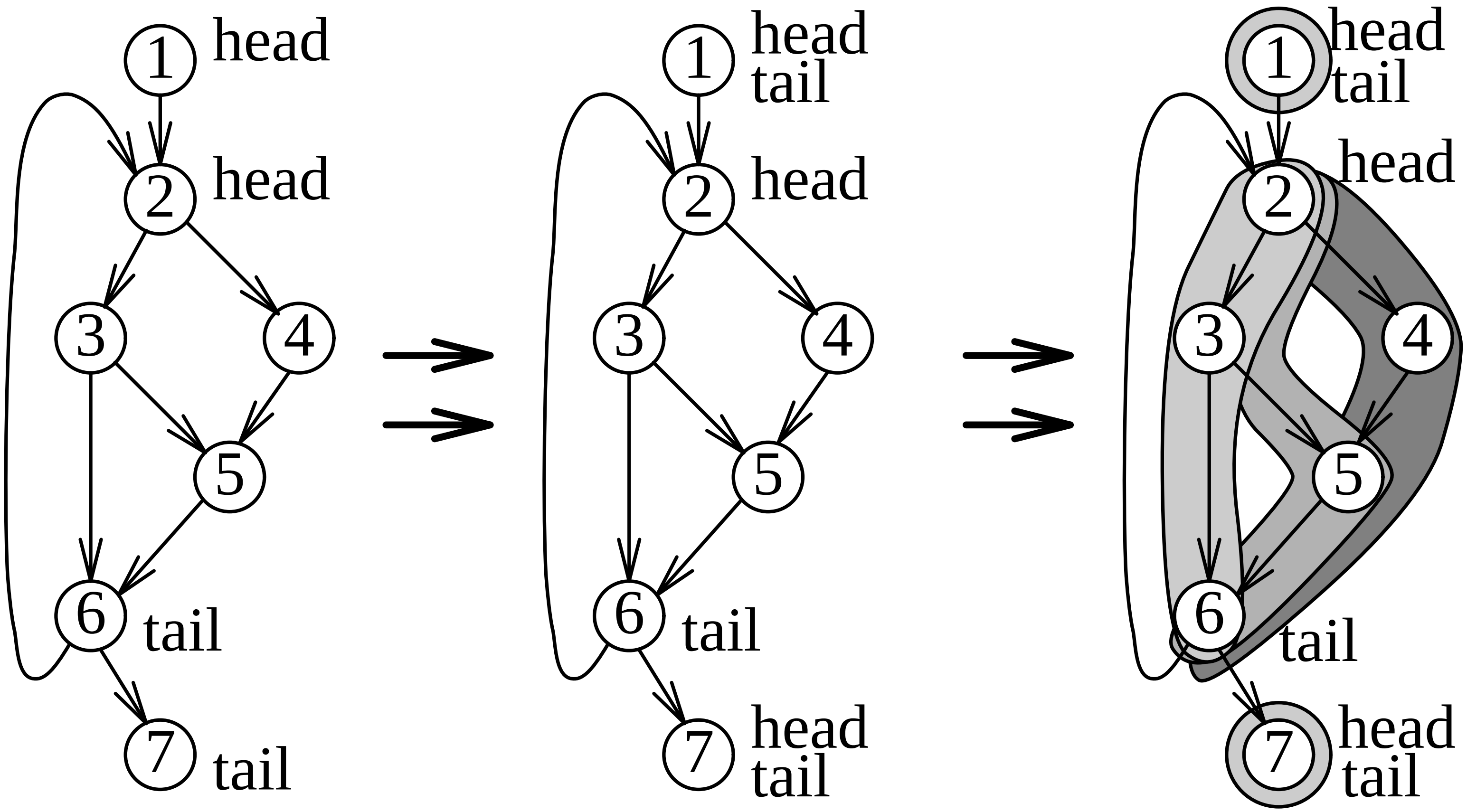
Computation of a Small UPPA

1. mark initial head and tail vertices
2. WHILE change DO
   (a) propagate heads and tails
   (b) for each new head vertex, find fork after join
3. UPPA = collect each path between a head and a tail vertex

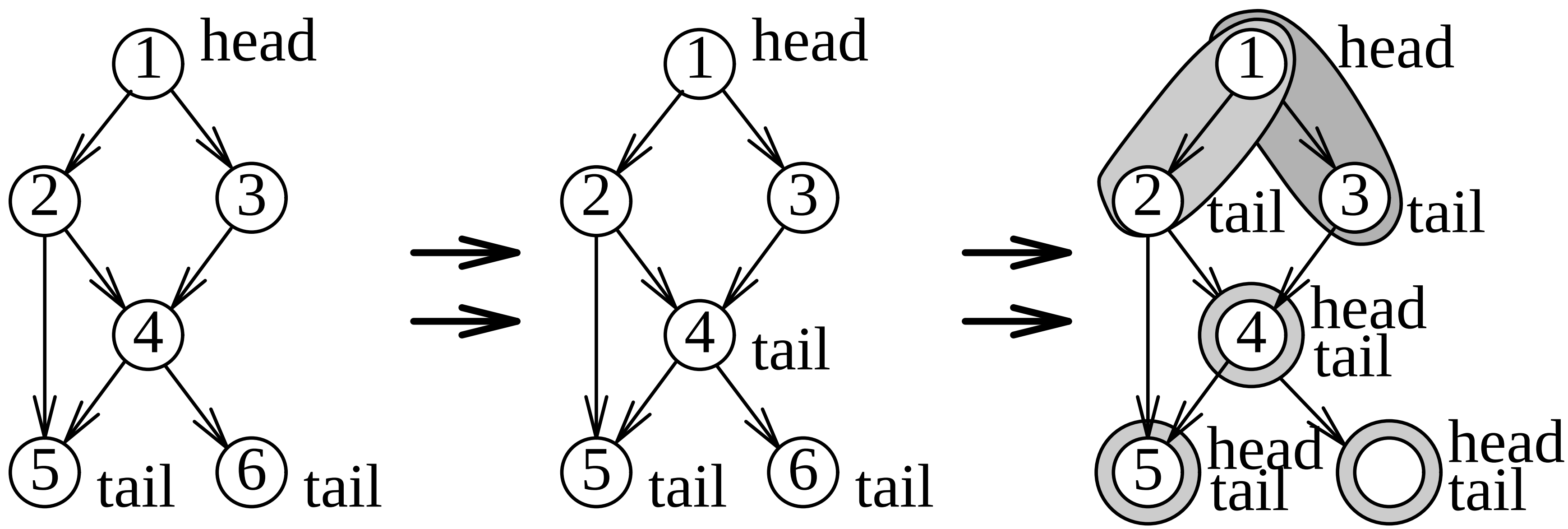
Example: Join followed by Fork



Example 1: Algorithmic Construction of a Small UPPA



Example 2: Algorithmic Construction of a Small UPPA



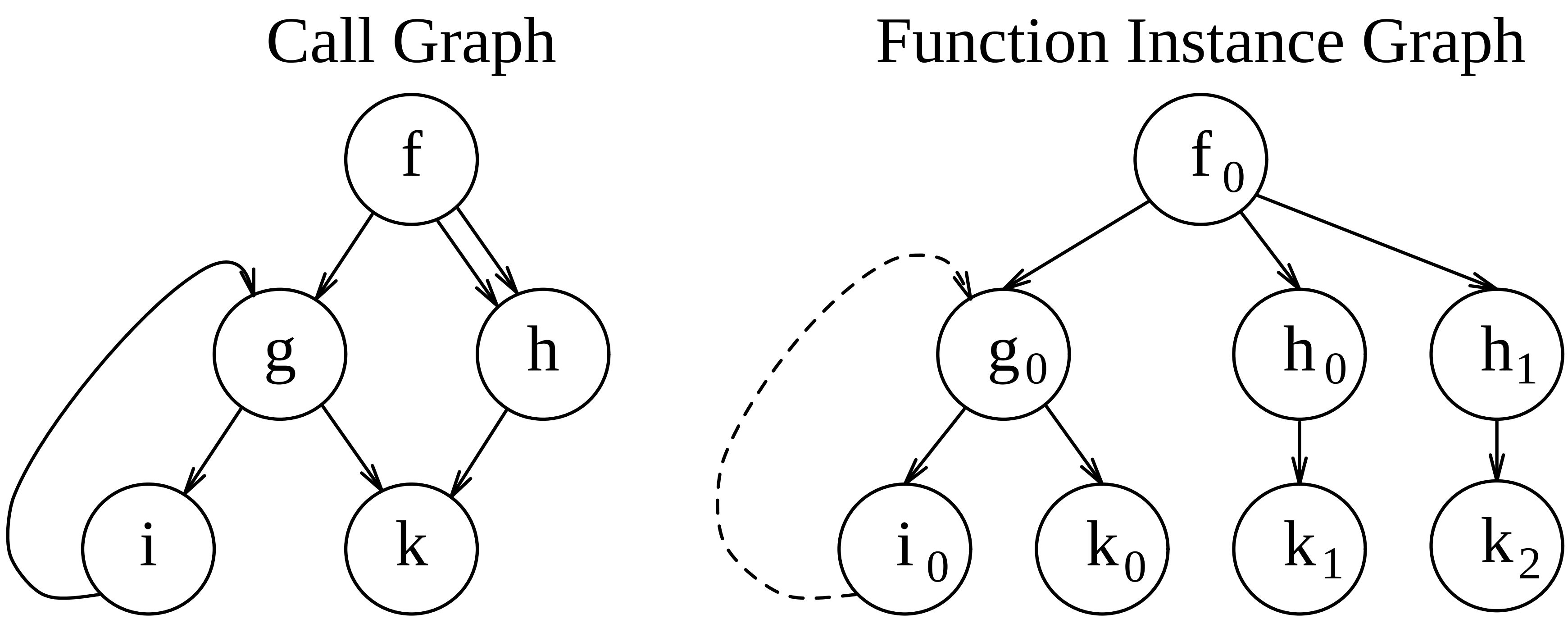
Properties of the Algorithm

- correctness proved
- minimal UPPA $\Rightarrow$ optimal on-the-fly analysis ??
  - define equivalence class of same-order UPPAs
  - show that algorithm constructs one such UPPA
  - show that no smaller UPPAs exist

Function-Instance Graph

- decomposition of call graph according to call sites
- useful for inter-procedural analysis in general
- used here for static cache simulation
- provides more detailed information about a function instance
- many applications: alias analysis, caller-save, inlining
- special transitions to recognize recursion

Example of Function-Instance Graph



Performance Evaluation

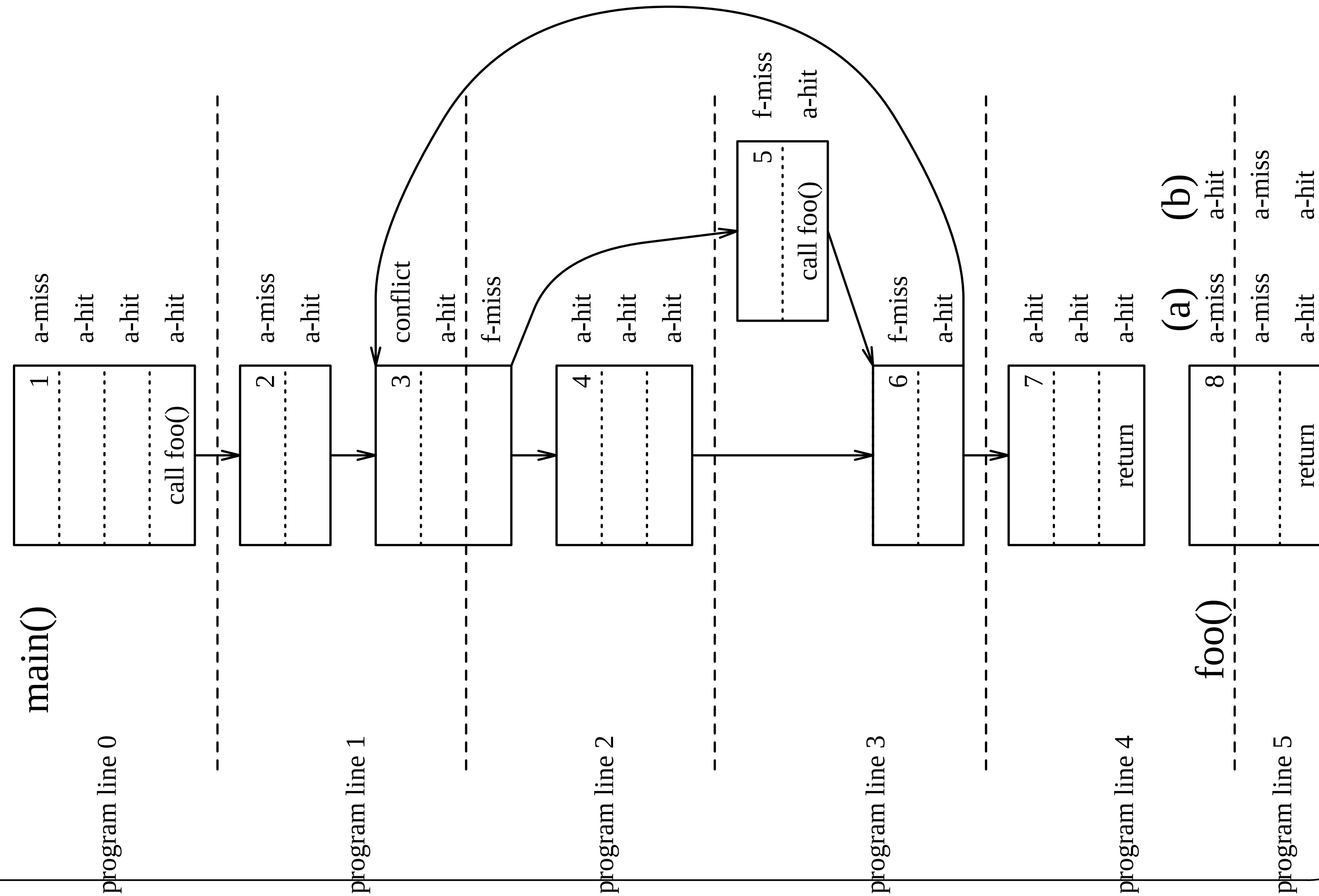
- UPPAs and function instances *vs.* basic block partitioning
- static savings: 24% fewer measurement points
- dynamic savings: 31% fewer measurement points

What is Static Cache Simulation?

- new approach to analyze cache behavior of programs statically
- applied to instruction caches (working on data caches)
- addresses of instructions known statically
- uses data-flow analysis of call graph and control flow
- categorizes each instruction
- predicts large portion of instruction cache references

Instruction Categorization

- transforms call graph into function-instance graph (FIG)
- performs analysis on FIG and UPPAs
- uses data-flow analysis algorithms for prediction
- *abstract cache state*: potentially cached program lines
- *reaching state*: reachable program lines
- categories based on these states:
  - always hit
  - always miss
  - first miss: miss on first reference, hit on consecutive ones
  - conflict: either hit or miss (dynamic)


main()
foo()
1
call foo()
a-miss
a-hit
a-hit
a-hit
2
a-miss
a-hit
3
conflict
a-hit
f-miss
4
a-hit
a-hit
a-hit
5
call foo()
f-miss
a-hit
6
f-miss
a-hit
7
return
a-hit
a-hit
a-hit
8
return
(a)
a-miss
a-miss
a-hit
(b)
a-hit
a-miss
a-hit
program line 0
program line 1
program line 2
program line 3
program line 4
program line 5

- 4 cache lines
- 16 bytes per line (4 instructions)
- instances foo (a) block 8a and (b) block 8b
- 7(1): always hit, spacial locality
- 8b(1): always hit, temporal locality
- 3(3): first miss
- 5(1) and 6(1): group first miss
- 3(1): conflict with 8b(2) conditionally executed

Fast Instruction Cache Performance Analysis

- uses efficient on-the-fly analysis
- performs static instruction cache simulation
- instruments program
- provides accurate cache performance measurements
- instrumented program has only 1.2 to 2.2 execution overhead
- faster than any other cache analysis method published so far

Conclusion

- general framework for efficient on-the-fly analysis (path partitioning)
- static cache simulation: new way to analyze caching behavior
- function-instance graph
- faster instruction cache performance analysis
- other applications