CS263
Program Profiling

Profiling

- "the formal summary or analysis of data representing distinctive features or characteristics" (*American Heritage Dictionary)*
  - Program profiling: analyzing the execution characteristics of code to extract and summarize its behavior

- Offline vs online
  - Offline – Collection time doesn't matter
  - Online – Slow-down is an important factor

- Exhaustive vs sample-based
  - Sampling is estimation. The difference between a sampled profile and an exhaustive profile is the accuracy/error measure
  - Sampling is commonly used online where time matters
    - Can be used offline if inaccuracy can be tolerated

Why Profile?

- To characterize program behavior
  - Understand how programs behave
  - Guide tool, runtime, system (hw/sw) design
  - Program test generation

- To capture specific or unusual behavior
  - Security attacks, intrusion detection, bugs, test coverage
  - Logging

Why Profile?

- To improve performance, track performance regressions
  - Time different parts of the program to find out where time is being spent
    - 80/20 rule – identify the 20 and focus your optimization energy
    - By hand optimization
    - Automatic (compiler or runtime) feedback-directed optimization
      - Target hot code
      - Inlining and unrolling
      - Code scheduling and register allocation
  - Increasingly important for speculative optimization
    - Hardware trends → simplicity & multiple contexts
    - Less speculation in hardware, more in software

What to Profile

- Individual instructions
  - Memory accesses (allocations/deletions, loads/stores)
    - Lends insight into caching, paging, garbage collection, bugs & more
  - If individual instruction detail isn't needed: capture basic blocks
    - Estimate bb's by recording branches and their direction
    - Lends insight into branch miss overhead
- Paths
- Function invocations and callsites
- Memory allocation, GC time
- Interfaces (ABI, APIs to other components, foreign function)
- Resource use
  - CPU, Network, disk, other I/O
  - Runtime services (compiler/interp, GC, runtime, OS)
- User interactivity

Instrumentation vs Event Monitoring

- Instrumentation: Insert code into the code of a program
  - The additional code executes interleaved with program code
  - To collect information about the program code activity
- Can perturb the behavior that it is trying to measure

Instrumentation vs Event Monitoring

- Instrumentation: Insert code into the code of a program
  - The additional code executes interleaved with program code
  - To collect information about the program code activity
- Event monitoring
  - Profiling external to the executing program
  - Output timestamps, upon OS or runtime activity, around program
  - Record of operations (timings, counts) in runtime that execute concurrently with the executing program, yet independent of it
    - Garbage collection activity
    - Accesses to the OS
    - Accesses to libraries (e.g. GUI)
  - Hardware performance counters/monitors (HPMs)
- Can perturb the behavior that it is trying to measure

Adaptive Optimization

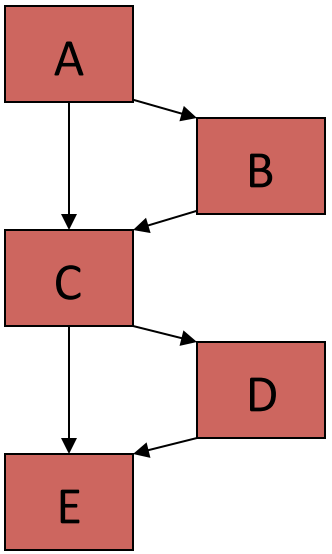
- Sample the system (lightweight)
- Predict future behavior based on past behavior
  - Does the past predict the future?
- Determine if prediction can amortize the cost of applying more optimization overhead

- Sampling to find hotspots or problem methods
  - Periodically record the top N methods on the runtime stack
    - Finding the right period and a value for N is tricky!
  - Use HPMs to identify methods that are causing stalls in the hardware… Careful, calling HPM services increments counters
    - Branch mispredictions
    - Cache misses
  - Very low overhead (< 2%)

Exhaustive Path Profiling (Instrumentation)

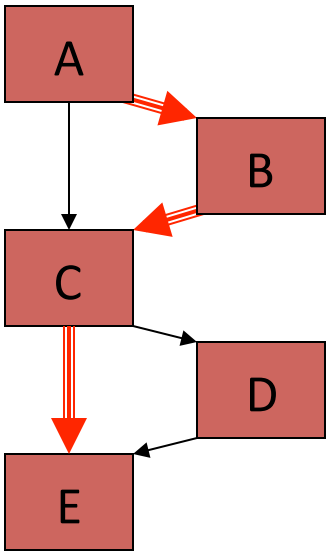
Thanks to Mike Bond (Ohio State) for his presentations of PEP and Continuous Path/Edge Profiling for these slides [CGO/MICRO 2005] on path profiling and its optimization.

- Processors need long instruction sequences
- Programs have branches



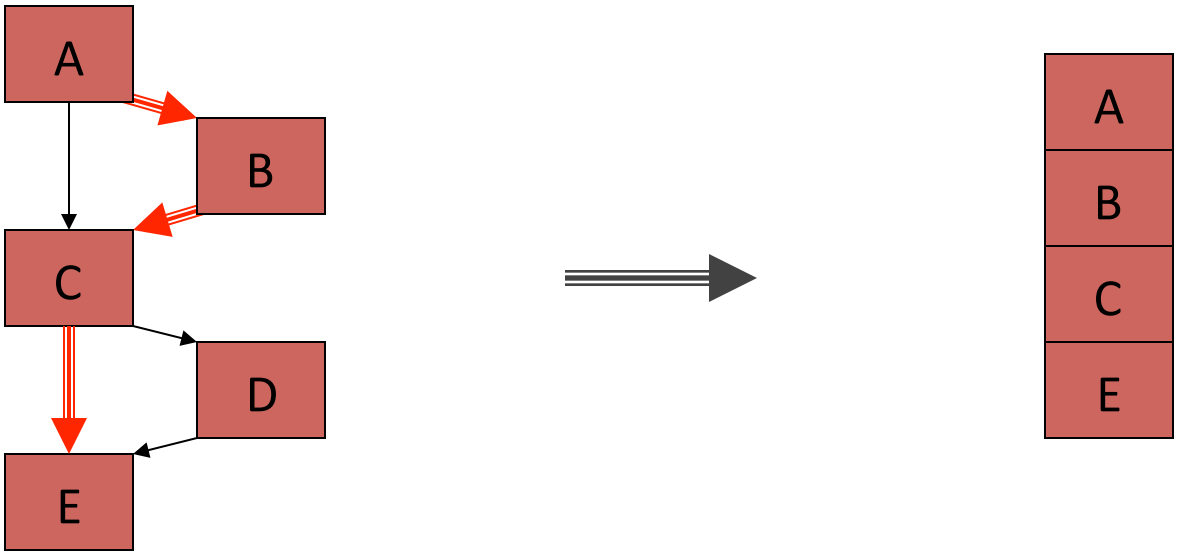
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks



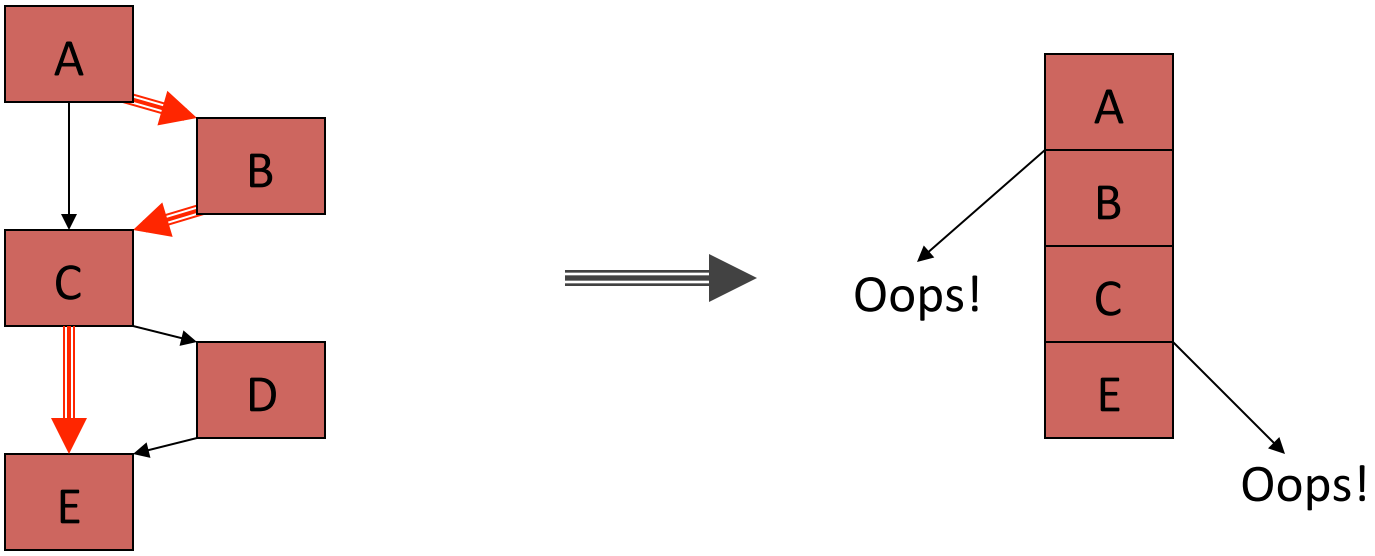
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Forms and optimizes "traces"



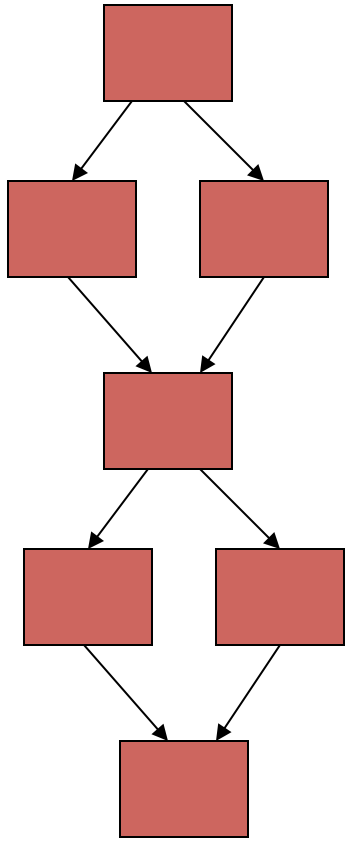
Why path profiling?

- Compiler identifies hot paths across multiple basic blocks
  - Forms and optimizes "traces"



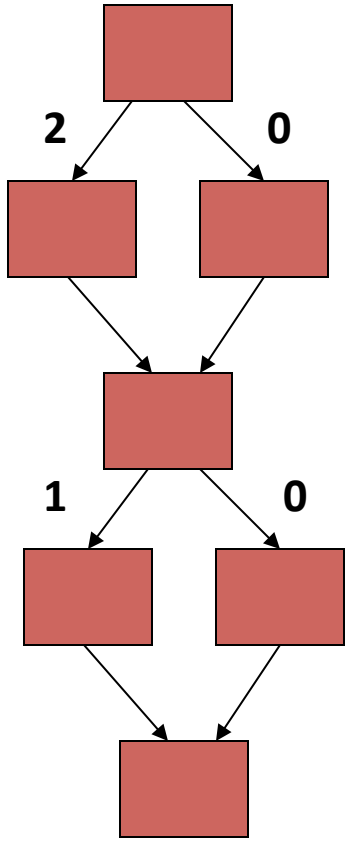
Ball-Larus path profiling

- 4 paths → **[0, 3]**



Ball-Larus path profiling

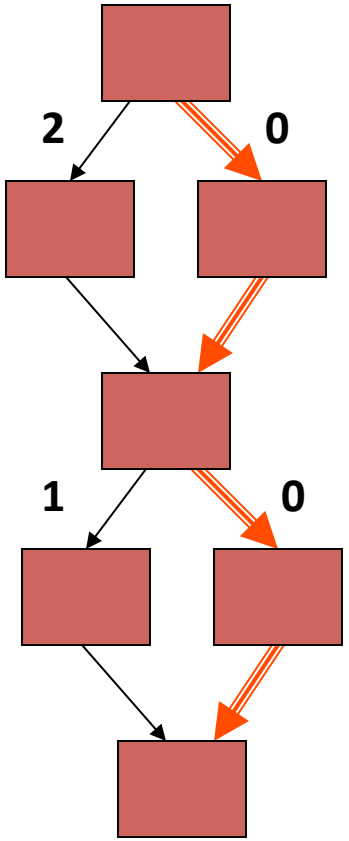
- 4 paths → **[0, 3]**
- Each path sums to unique integer



Ball-Larus path profiling

- 4 paths → **[0, 3]**
- Each path sums to unique integer

Path 0

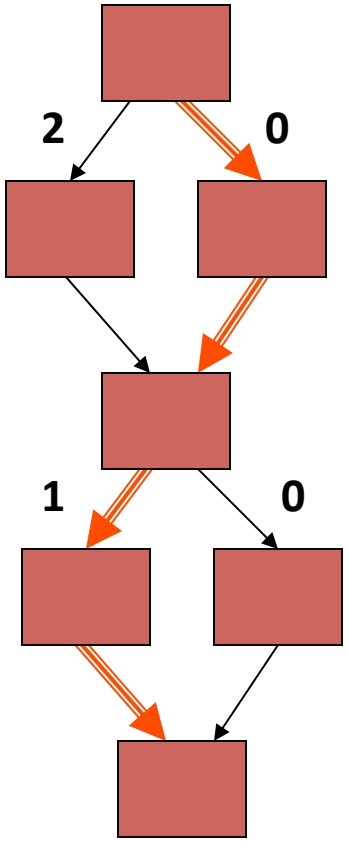


Ball-Larus path profiling

- 4 paths → **[0, 3]**
- Each path sums to unique integer

Path 0

Path 1



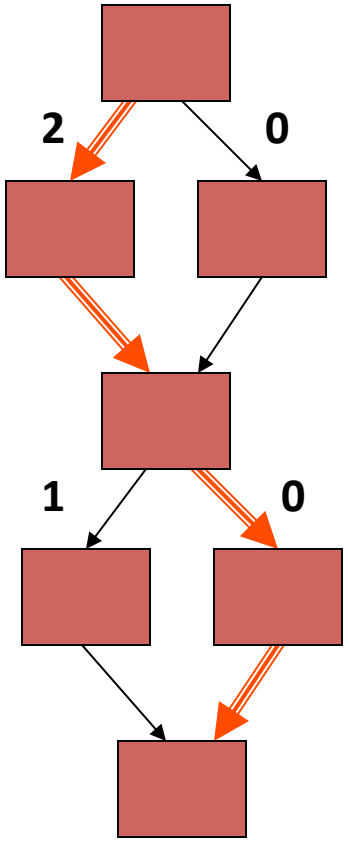
Ball-Larus path profiling

- 4 paths → **[0, 3]**
- Each path sums to unique integer

Path 0

Path 1

Path 2



Ball-Larus path profiling

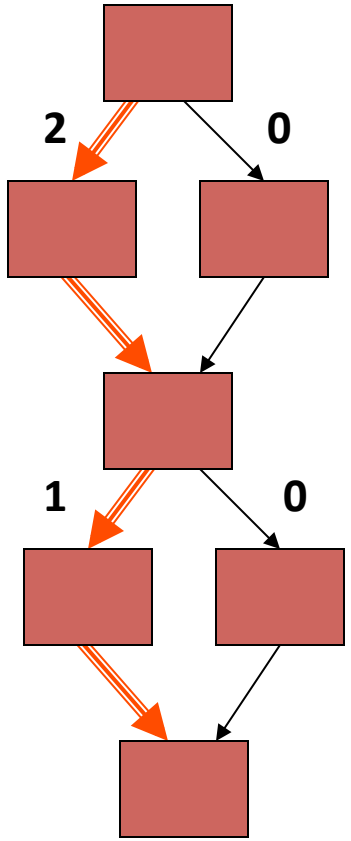
- 4 paths → **[0, 3]**
- Each path sums to unique integer

Path 0

Path 1

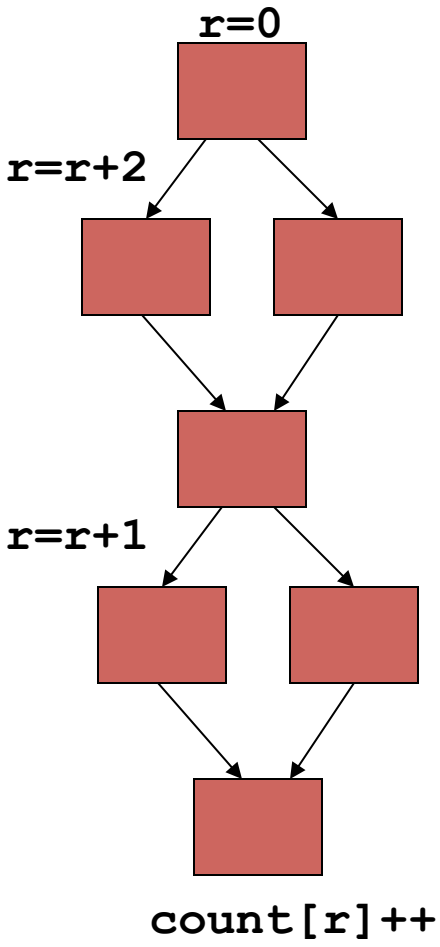
Path 2

Path 3



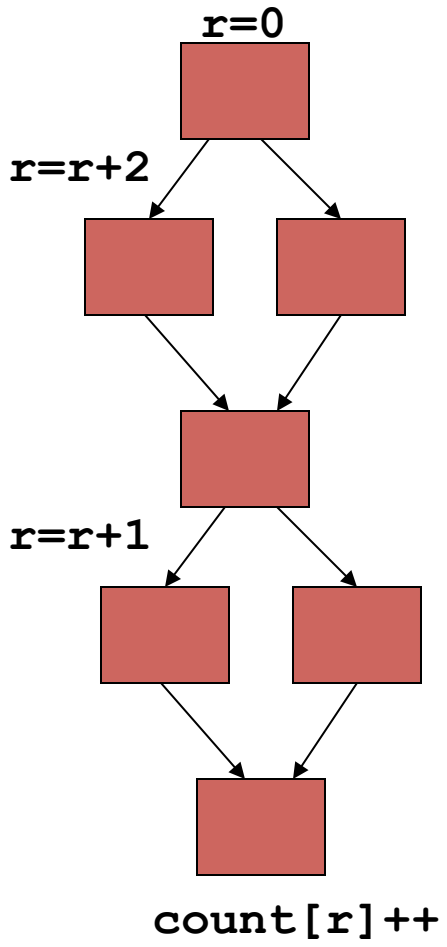
Ball-Larus path profiling

- **`r`**: path register
  - Computes path number
- **`count`**:
  - Stores path frequencies

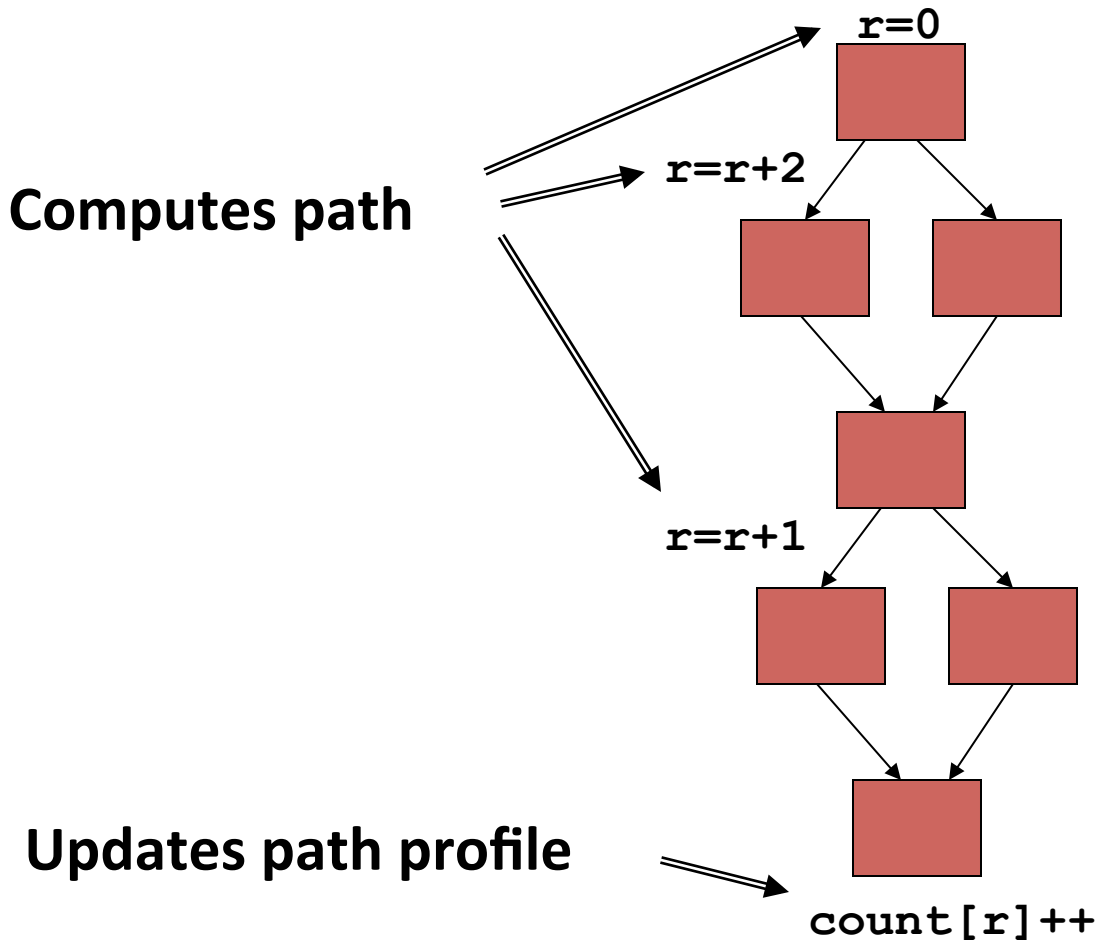


Ball-Larus path profiling

- **`r`**: path register
  - Computes path number
- **`count`**:
  - Stores path frequencies
  - Array by default
  - Too many paths?
    - Hash table
    - High overhead

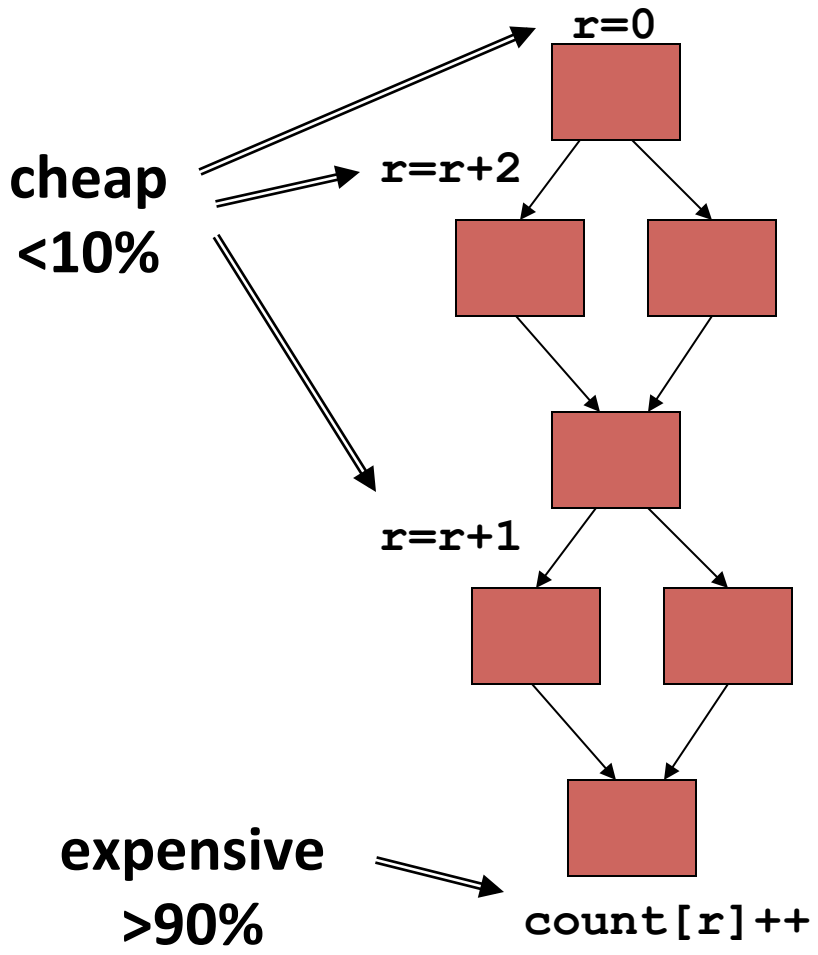


Optimizing Path Profiling

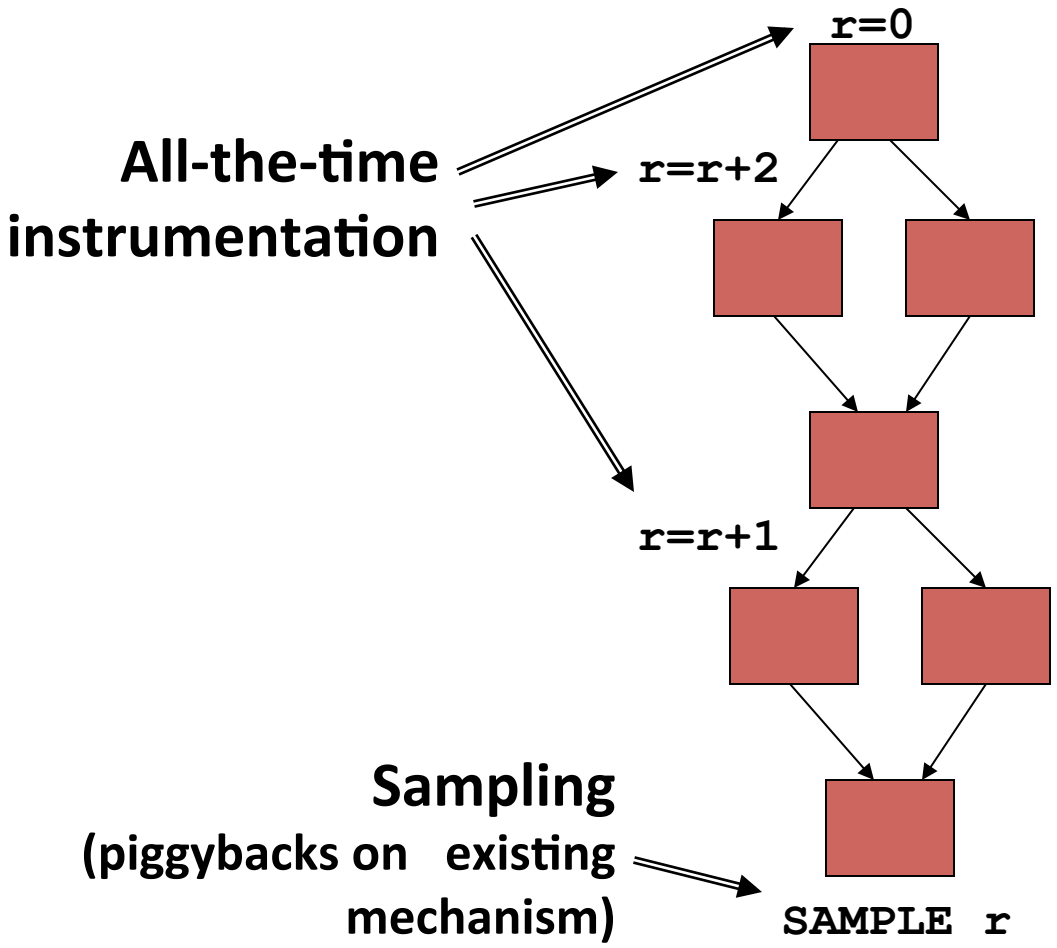


Optimizing Path Profiling

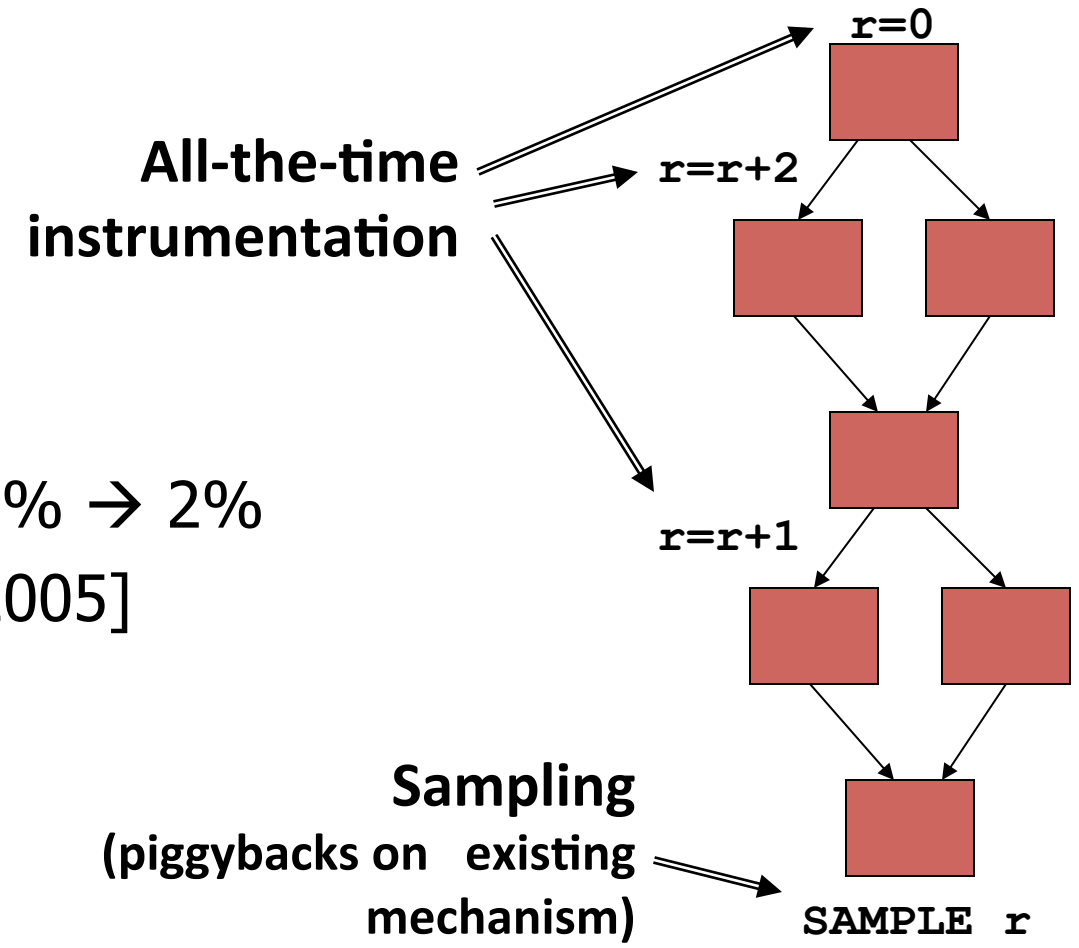
- Where have all the cycles gone?



Optimizing Path Profiling



Optimizing Path Profiling

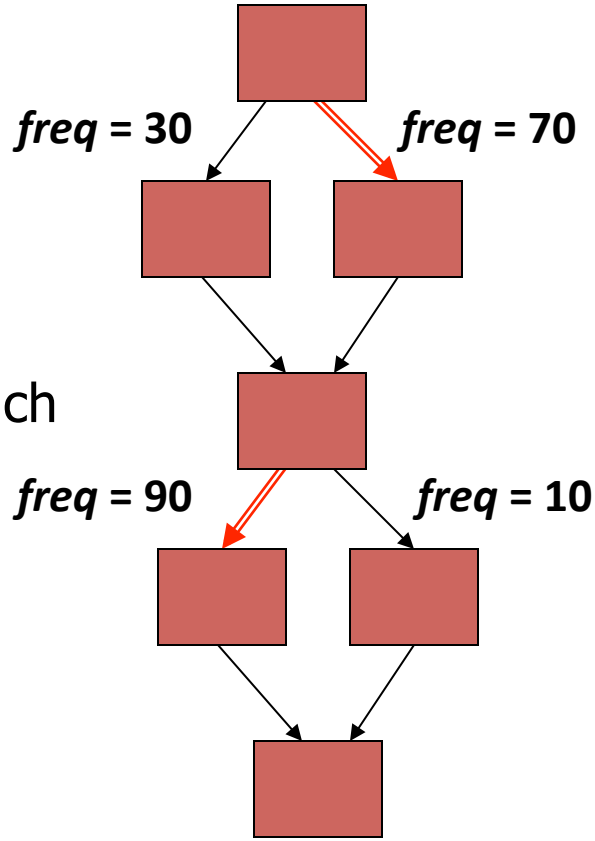


Overhead: 30% → 2%

[Bond et al. 2005]

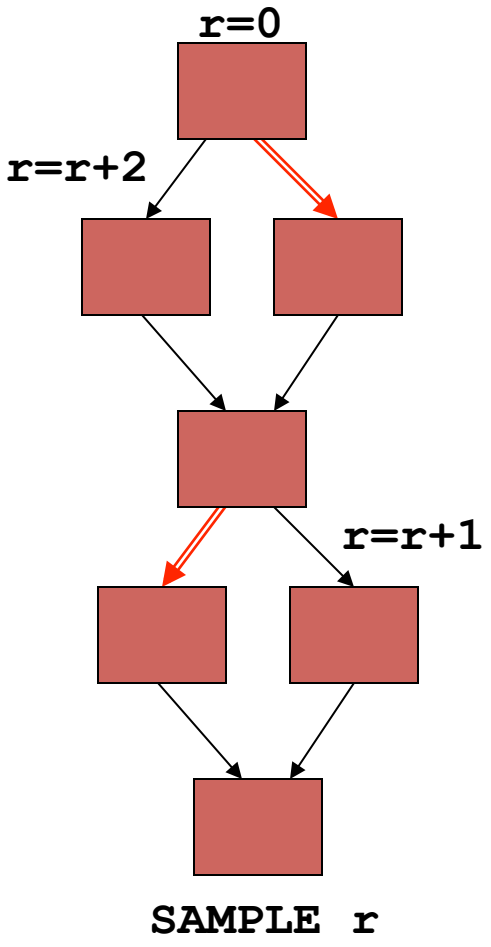
Profile-guided profiling

- Existing edge profile informs path profiling
  - Profile some initially
    - Quite fast to profile edges
    - Can be sample based
    - Just need to determine which branch edges are taken more frequently



Profile-guided profiling

- Existing edge profile informs path profiling
- Assign zero to hotter edges
  - No instrumentation



Sample-based Instrumentation

- Turn on and off instrumentation dynamically
  - Challenge: when to turn instrumentation on and off
  - Why is this important to do?
  - How:
    - Via code patching: Ephemeral Instrumentation, DynInst, IBM Java Developer Kit
      - Have two versions of the methods (or code blocks) you want to instrument
      - In the uninstrumented version, put a patch point at entry
        - » Dummy instruction large enough to hold a jump
      - Overwrite (patch) the entry point to instrument
      - "Undo" patch to turn off instrumentation
    - Via recompilation and on-stack replacement
  - Via code copying (today's paper)

Today's paper

- Summarize it
- Ways to turn instrumentation off (what's wrong with these?):
  - Patching
  - Continuously recompiling
  - OSR

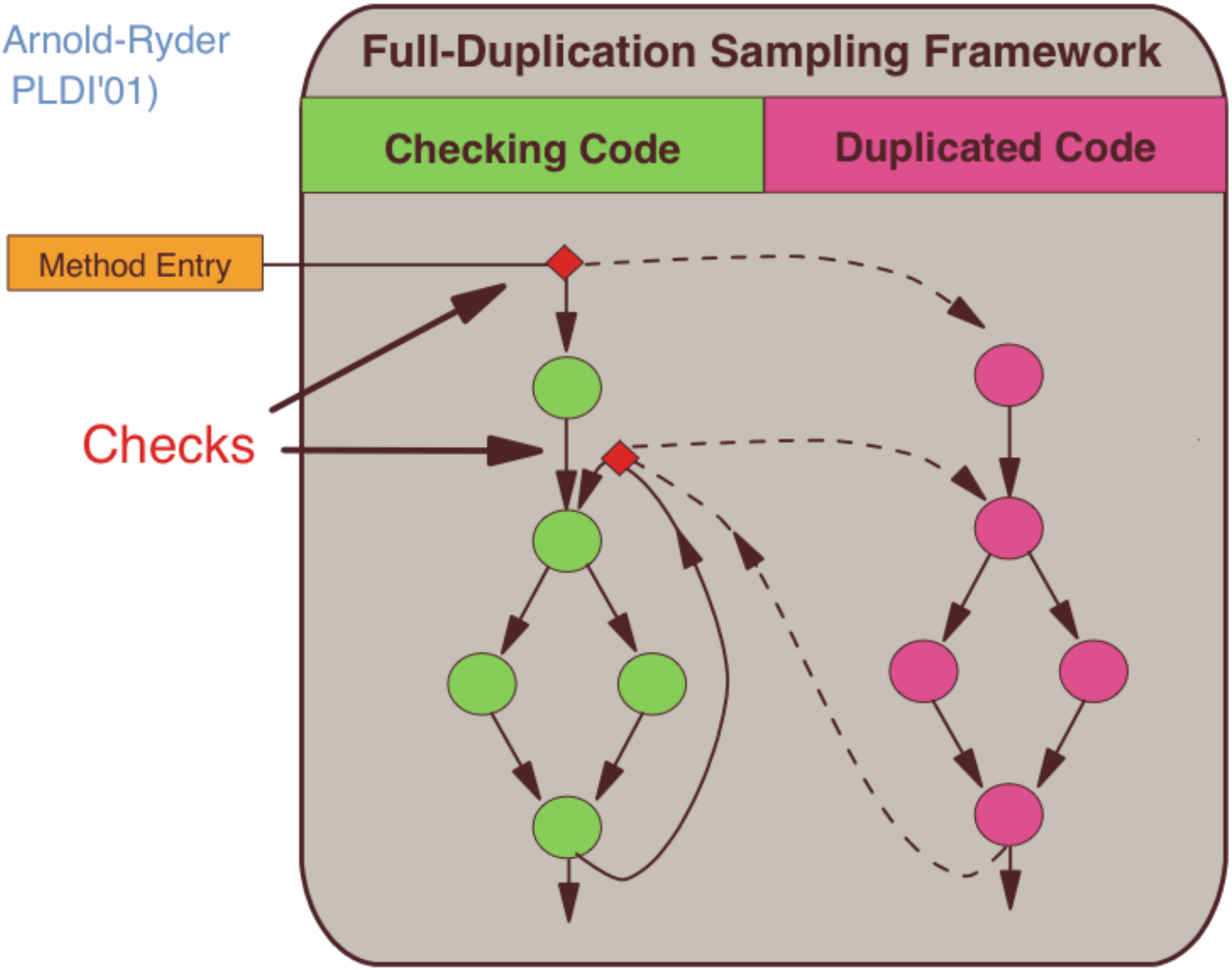
Today's Quiz

1. 2 advantages to this approach
2. What is checking code and what is its overhead
3. What is duplicated code and what is its overhead
4. Profile types used in experimentation
5. Disadvantages to the approach (2)
   - Solutions to one of the disadvantages

Today's Quiz

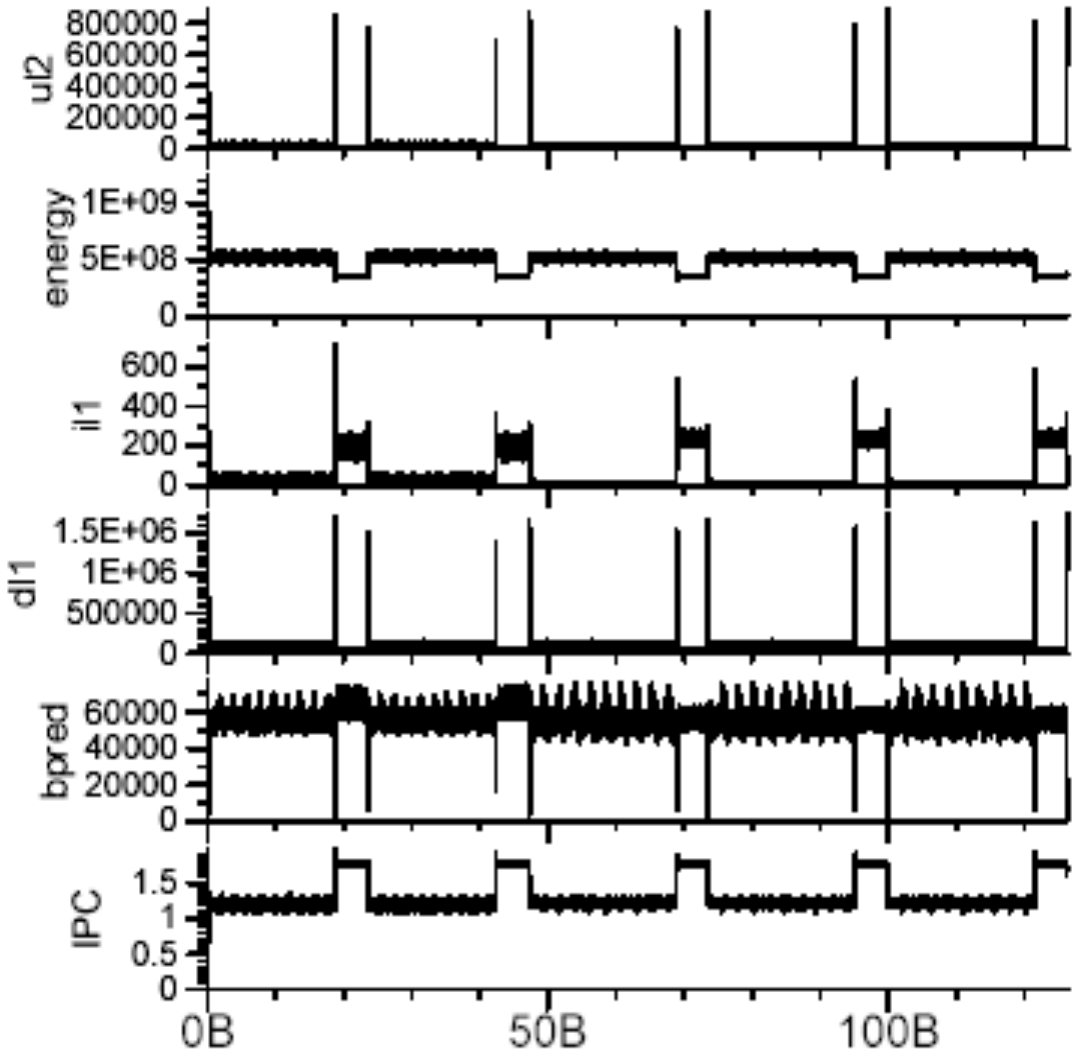
1. Advantages to this approach
   - Low overhead, high accuracy sampling
   - Simple
   - Controllable
2. What is checking code and what is its overhead
   - Checks on back edges and method entry (on all of the time)
     - To determine when to switch to instrumented code
3. What is duplicated code and what is its overhead
   - Instrumentation
   - Space for instrumented code copy
4. Profile types: call edge and field accesses
5. Disadvantages: space/time overhead
   - Checking, duplicated code, compile time
   - Code bloat solutions: Partial duplication and no duplication

(Arnold-Ryder PLDI'01)



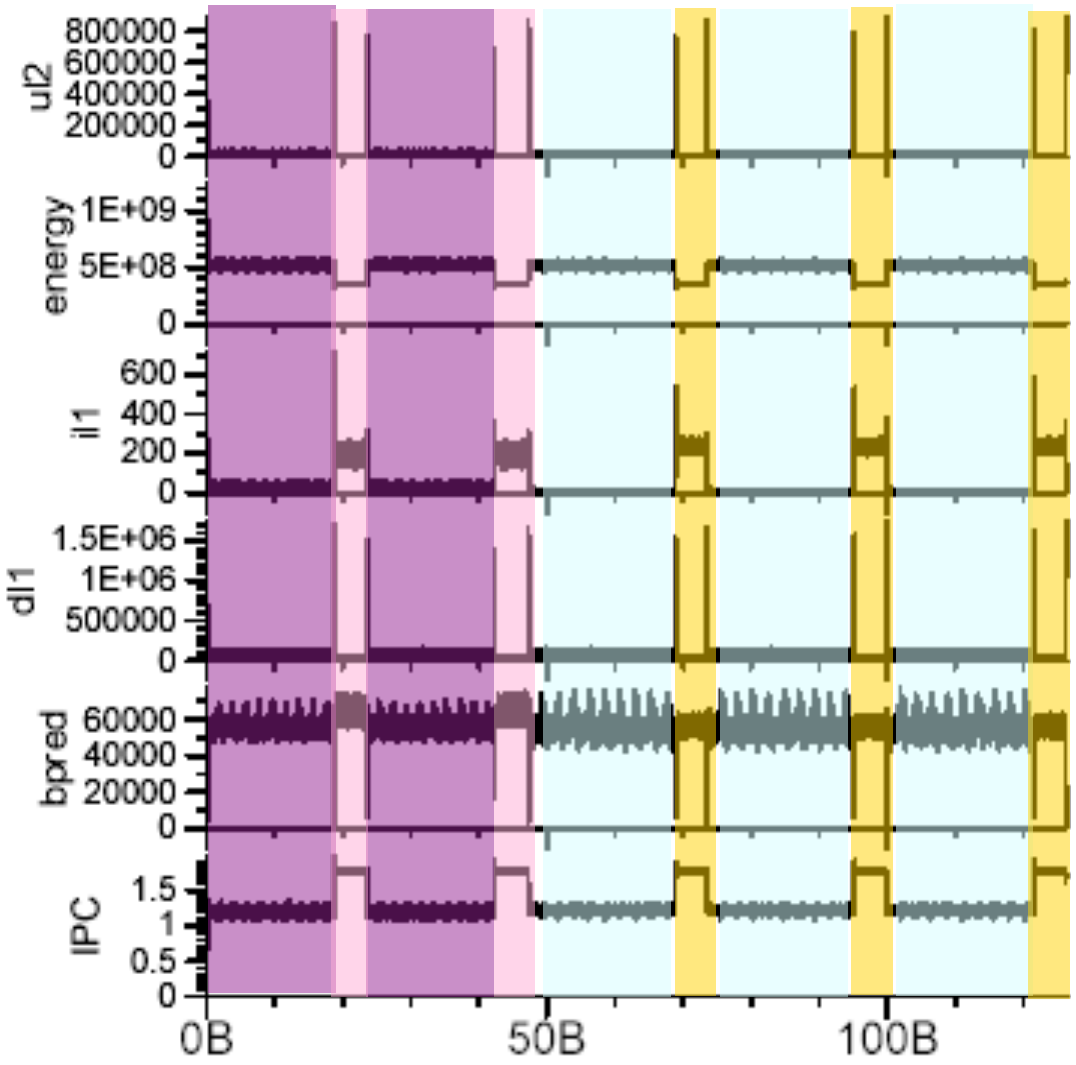
Hardware Performance Monitors/Counters

- Libraries provide access to hardware collected HPMs
- Other types of sampling
  - Random
  - Periodic
  - Phase



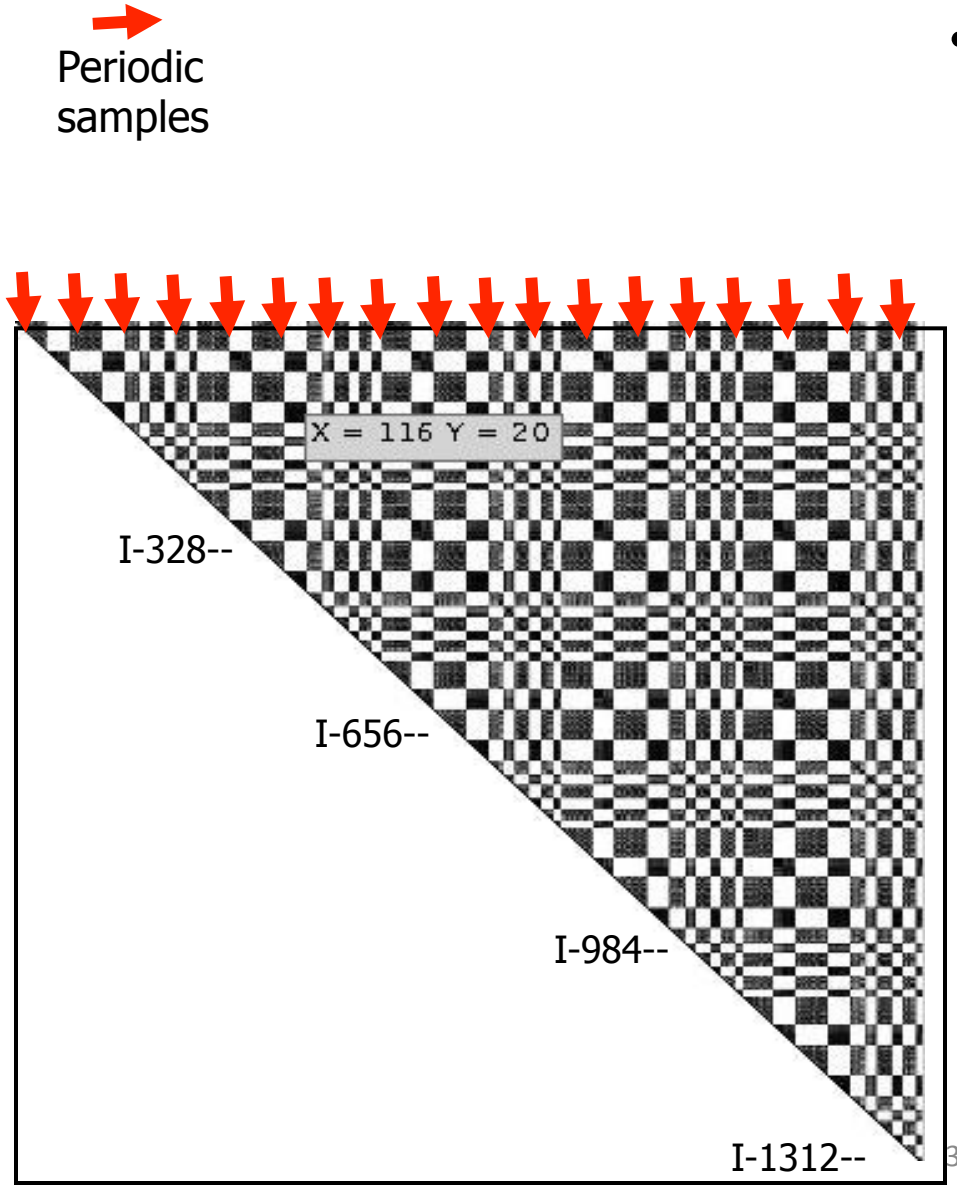
Time Varying Behavior of Programs

Program Behavior changes over time



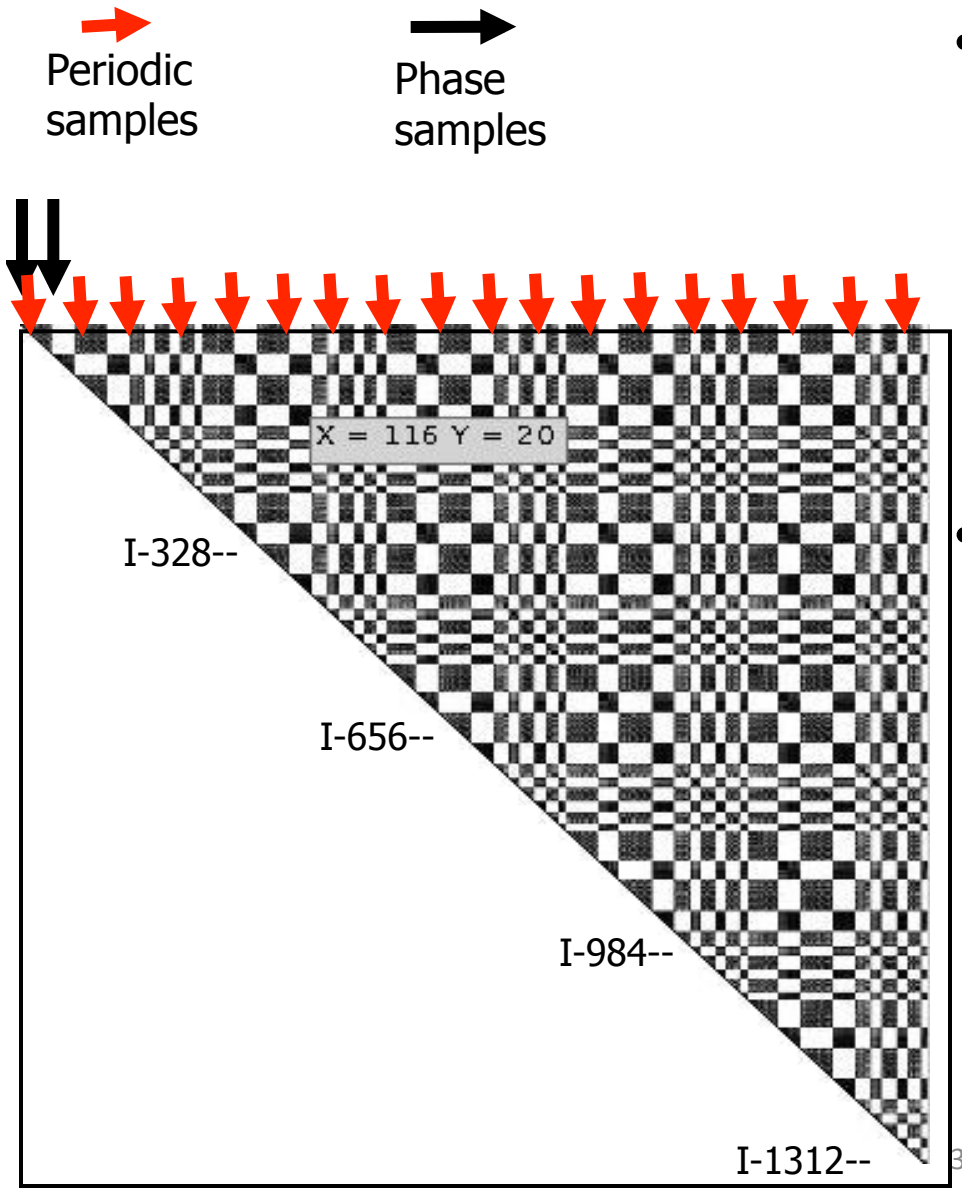
- Different behavior during different parts of execution
- Many programs execute as a series of phases possibly with **recurring** patterns
- Capture via basic block profiles for fixed number of instructions=vector
  - Compare counts across vectors for similarity

Phase Aware Remote Profiling



- Extant approaches (random/ periodic sampling)
  - Do not consider time-varying and repeating behavior
  - Collect redundant information

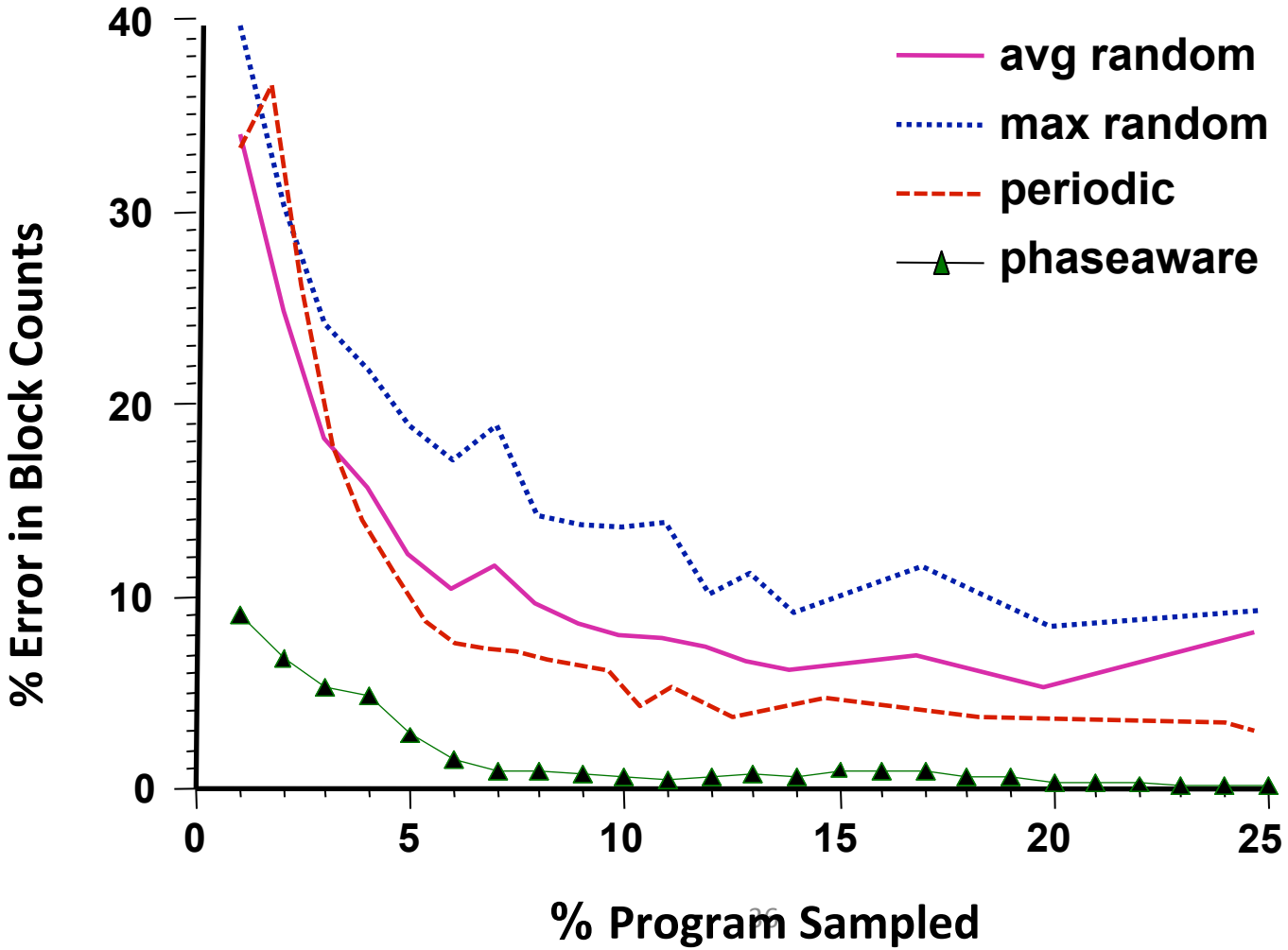
Phase Aware Remote Profiling



- Extant approaches (random/ periodic sampling)
  - Do not consider time-varying and repeating behavior
  - Collect redundant information
- Our approach: Sample according to **phase** behavior

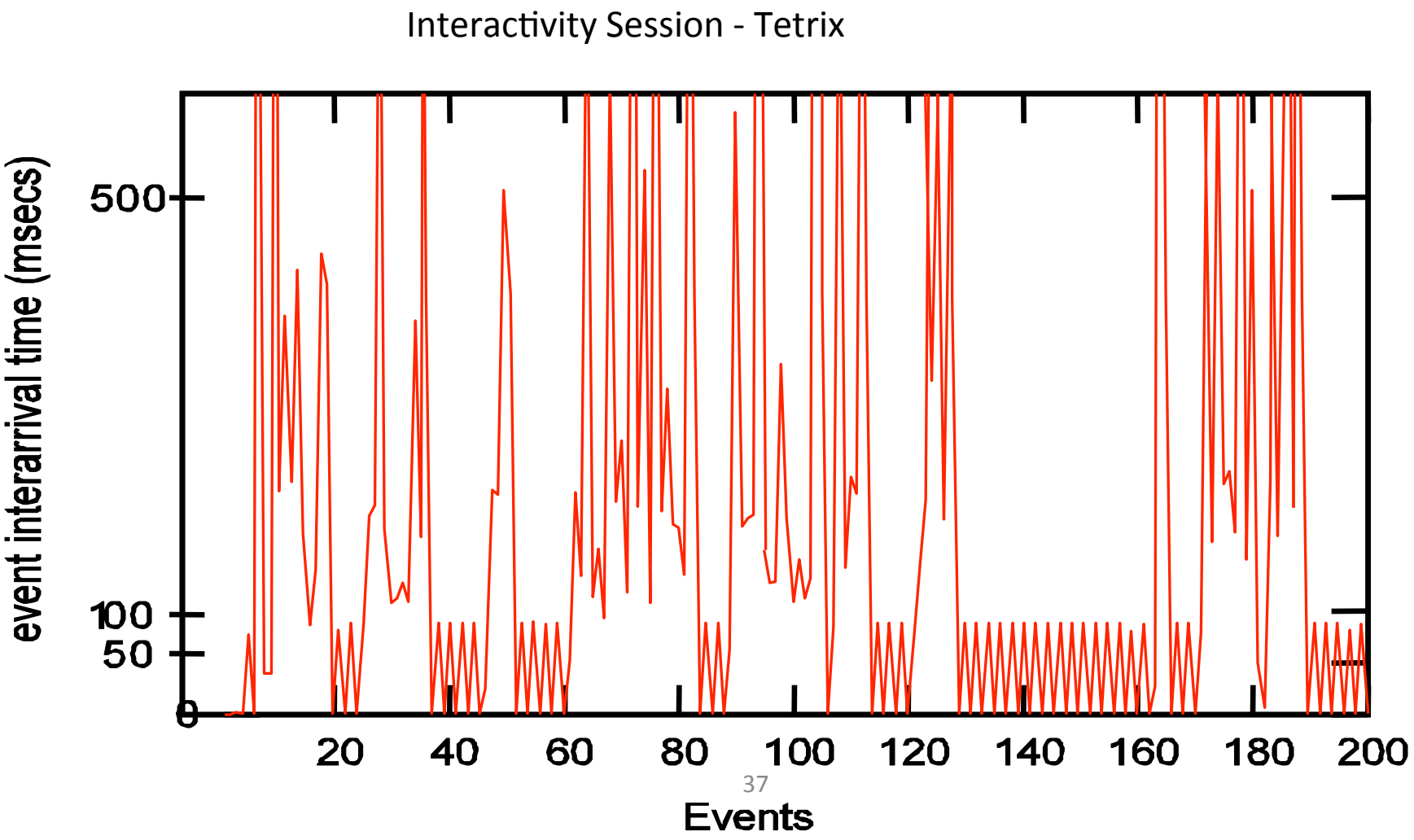
Results

- 50-75% reduction in overhead (over periodic and random sampling)



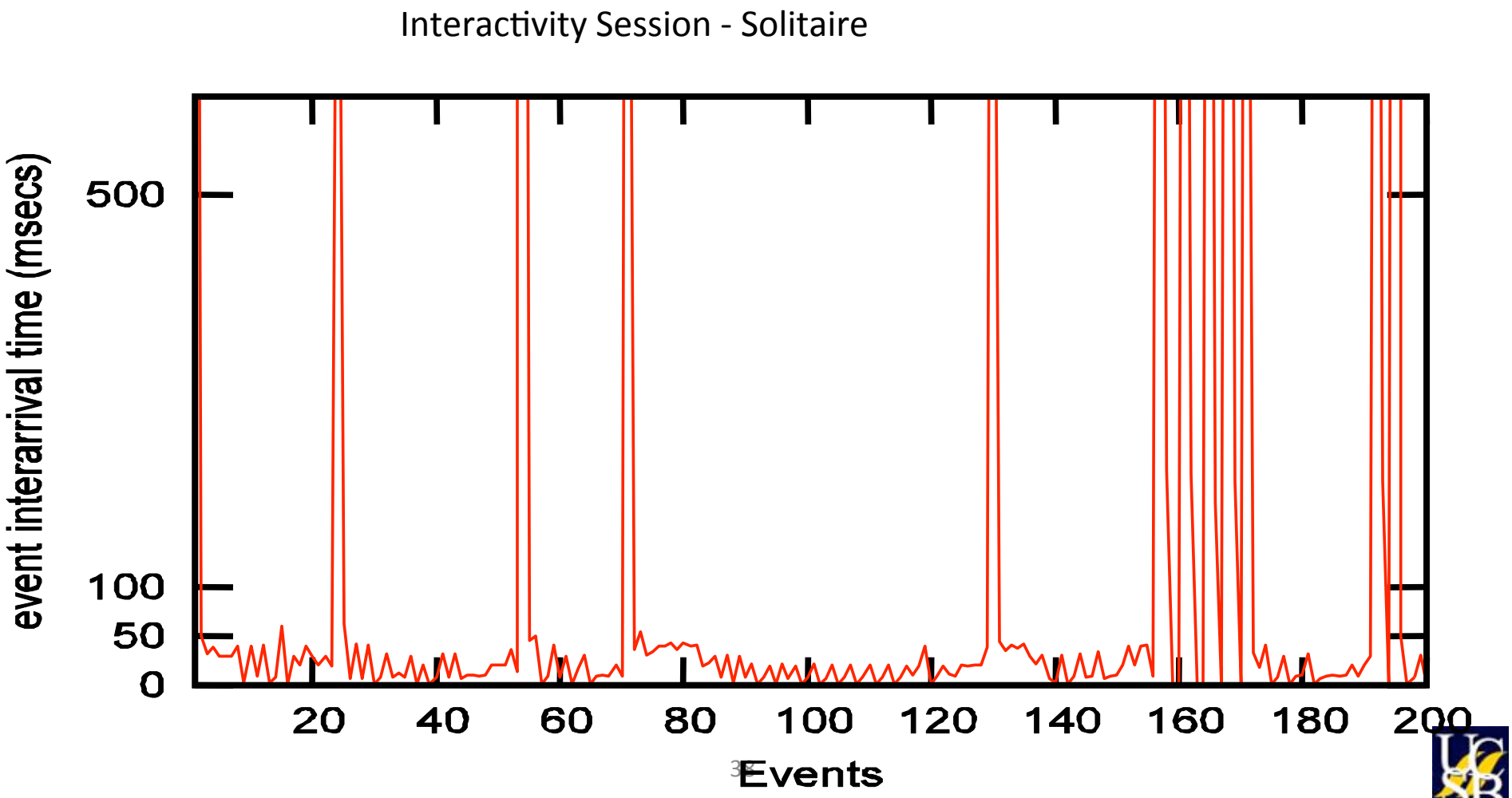
Sampling Interactive Sessions

- A period of user interaction: Each application has a specific pattern



Interactive Sessions

- A period of user interaction: Each application has a specific pattern



Profiling Tools

- Of binaries (independent of language)
  - Pin, Dynamo
  - Valgrind
  - gprof (call graph and function timings)

- Of programs (language specific)
  - Java – JVMPI/TI, JProfiler, many others, GCSpy
  - Ruby – ruby-prof
  - Python – cprofile

- HPMs
  - Library support: PAPI
  - OS Integration: PerfMon, OProfile, XenOProf