Efficient Tracing of Cold Code via Bias-Free Sampling

Baris Kasikci[+], Thomas Ball[*],
George Candea[+], John Erickson[*],
Madanlal Musuvathi[*]

[+]ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

[*]Microsoft Research

Why Should We Sample Cold Code?

- Cold code is not well tested
  - *Bugs lurk in cold code [Marinescu et al., Cristian et al.]*
- Cold vs. hot code is not known a priori
  - *Cold code is rarely executed during program execution*

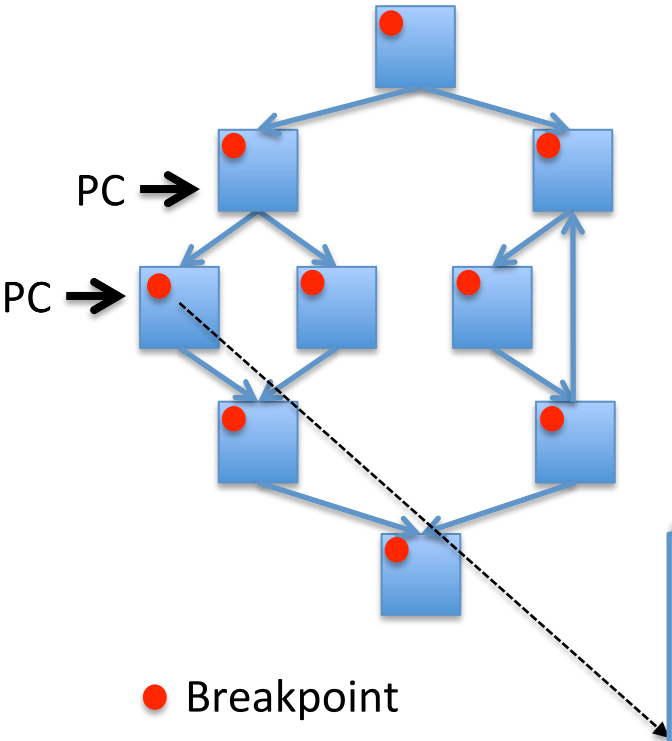
We need to be able to efficiently sample cold code

Current Dynamic Sampling Approaches

- Static instrumentation (e.g., Gcov, bbcover)
  - *Incurs lots of overhead (>2x)*
  - *Requires separate builds*
- Dynamic instrumentation (e.g., Pin-based)
  - *Do not handle multithreaded programs efficiently*
- Temporal sampling (e.g., CBI [Liblit et al.])
  - *Less overhead per-execution*
  - *Need lots of executions to catch cold code*

Current approaches are inefficient and do not scale

How to Efficiently Sample Cold Code?



- Use code breakpoints
  - One breakpoint per basic block
  - Present in all modern CPUs
  - 0 cost once removed

- Sample instruction
  - Mark as “executed”
  - Record the accessed memory address
  - …

Challenges

- Don’t change behavior of
  - *Instrumented programs*
  - *Services such as debuggers*
- Number of breakpoints
  - *In the worst case, a breakpoint for every block*
  - *Existing frameworks cannot handle such volume*
- Multithreaded code
- JIT and managed code
  - *Cannot be handled like normal code due to optimization*

Bias-Free Sampling (BfS)

- Design
- Implementation
- Evaluation

Native/managed, kernel/user space, x86/ARM
Ran on 679 programs, incurs overheads of 1-6%

BfS’s Design Goal

- Sample cold instructions without over-sampling hot instructions
- Sample all the other instructions independently of their execution frequency

```
for (i=0; i<1,000,000; ++i)
  if (…)
    statement_1
  else
    statement_2
```

Executes once every one million iterations

BfS Parameters - Definitions

- K: Desired sample count per-instruction
  - *Ensures first K executions are sampled*
  - *Bounds the overhead*
  - *0 cost after K breakpoints*
- P: Sampling distribution
  - *Can be uniform or biased*
- R: Sampling rate
  - *Number of samples generated per second*
  - *Controls the overhead*

BfS Parameters - Examples

Application	Count (K)	Distribution (P)	Rate (R)

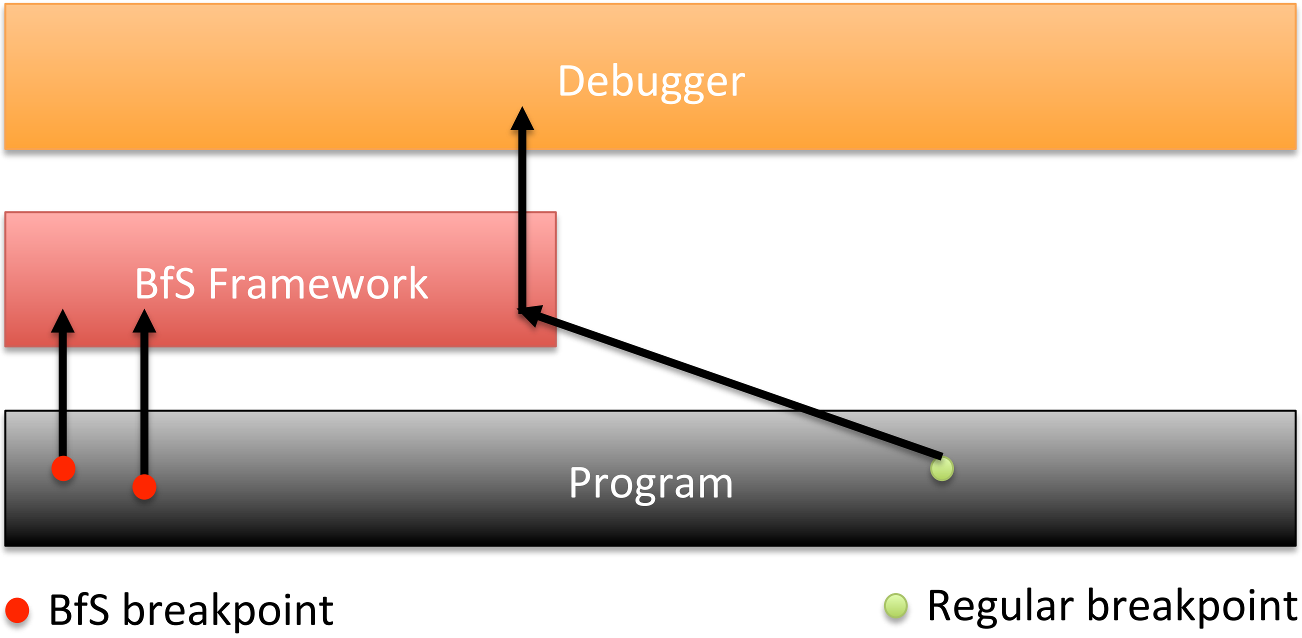
Bias-Free Sampling

- Design
- Implementation
- Evaluation

Breakpoints Primer

- Hardware support
  - `int 3` *on x86 traps into the OS*
- Breakpoint instructions are not larger than any instruction in the ISA
  - *Allows overwriting only a single instruction*
  - *Atomic add/removal*
  - *Helps lower the overhead*

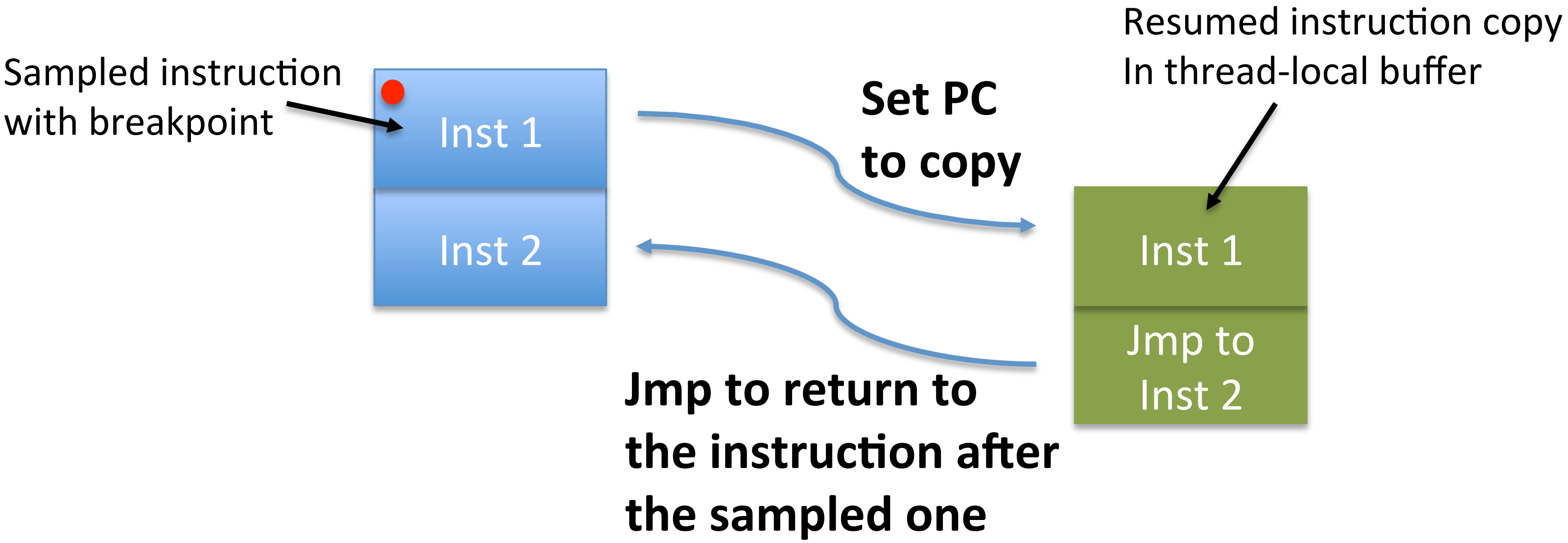
Debugger Interplay



BfS framework is invisible to the debugger, allowing transparent breakpoint processing

Multi-Shot Breakpoints

- Debuggers processing a breakpoint
  - *Restore original instruction*
  - *Single step*
  - *Restore the breakpoint*
- BfS framework



Sampled instruction with breakpoint

Set PC to copy

Resumed instruction copy In thread-local buffer

Jmp to return to the instruction after the sampled one

Managed Code Support

- BfS uses CLR debugging APIs
  - *Bypassing the APIs does not work*
  - *CLI (interpreter) performs introspection*
  - *Cannot modify the binary without the CLR's knowledge*
- May need to disable JIT optimizations for some tasks
  - *E.g., to have exact coverage results*

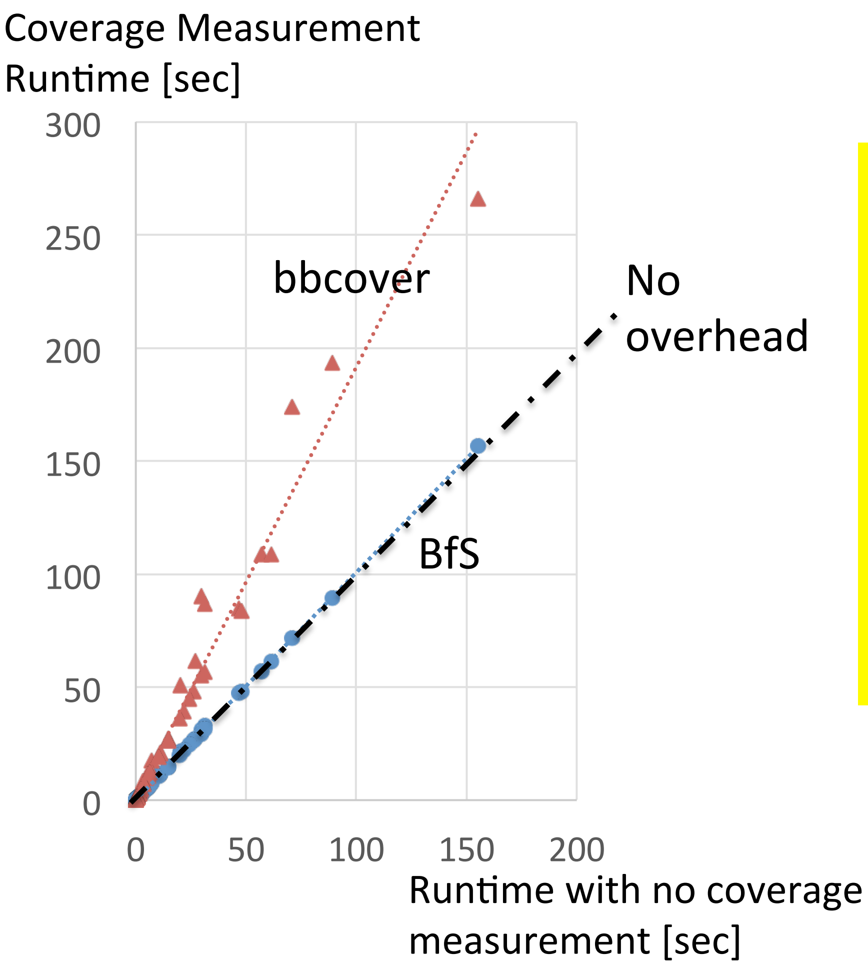
Bias-Free Sampling

- Design
- Implementation
- Evaluation

<u>679 programs:</u>

All Windows system binaries, Z3 constraint solver, SPECint benchmark suite, and C# benchmarks

Use Case 1 – Z3 Coverage



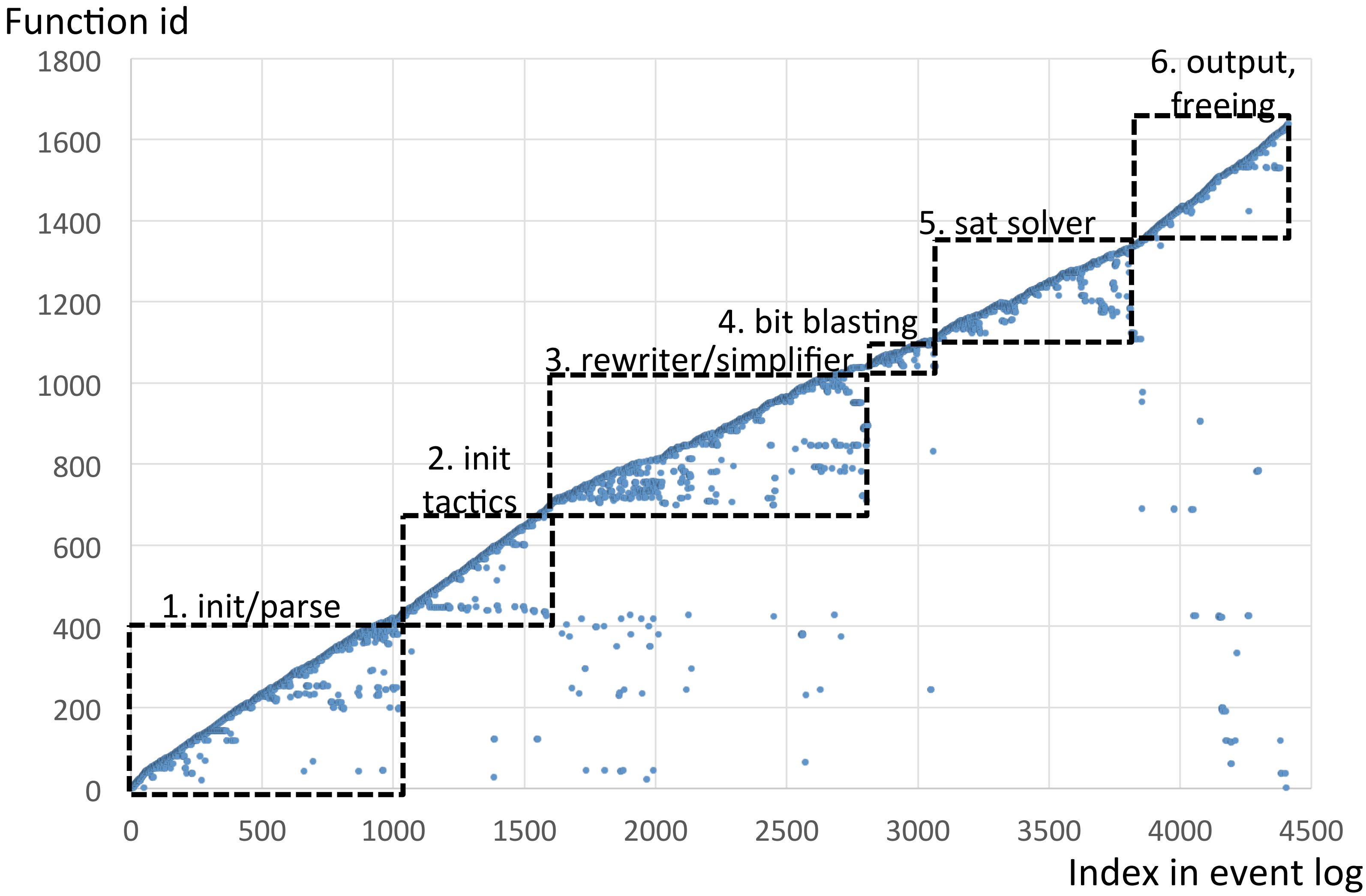
BfS’s coverage overhead (1%) is independent of program behavior, it is a function of program size

Use Case 2 – Coverage in Testing Windows 8 Binaries

- Coverage with BfS and bbcover
  - *665 system binaries: 32 and 64 bit, x86 and ARM*
  - *70 to 1,000,000 basic blocks*
  - *A total of 4 hours on 17 machines*
- bbcover failed for 45 binaries due to timeout
- For all but 40 tests, BfS reports more coverage
  - *Less coverage cases are due to non-determinism*

Coverage overhead is always less than 6%

Use Case 3 – Z3 Cold Code Tracing



Cold-code tracing identifies sets of related functions

Bias-Free Sampling

- Low overhead technique to identify cold code
- Leverages breakpoint support
  - *Ideal for multithreaded code*
  - *No need for a separate build*
- Implementation on various platforms
  - *32 and 64 bit, x86 and ARM, kernel and user space, native and managed*
- Comprehensive evaluation
  - *1-6% overhead for coverage and cold block tracing*