The Long and (Un)Winding Road
Stackwalking in HPCToolkit

John Mellor-Crummey, Nathan Tallent, Mike Fagan,
Mark Krentel, Gabriel Marin
What We Do

- Performance Analysis via call-stack sampling
- No instrumentation
- Must work on highly optimized binary
The Setup

Sample Event Fires

Walk (unwind) Stack
Store the stack walk, updating counters
This Talk

- Focus on x86 architecture
  
  [rbp = frame pointer]

- Focus on Linux

- Will use unwind and (stack)walk interchangably
Unwinding the Stack

• main -> A -> B -> C
• for a given code address in a call chain, figure out return address
• return address found via a *recipe* specific to the code address

RA = *(sp + 15)    RA = *(bp + 10)
More Unwinding

- Determine the recipe for the address
- apply the recipe
- repeat until done
x86 Receipes in Highly Optimized Code

- `sp` (scratch `bp`, `bp` unchanged = “frameless”)
- `bp` (imperative with dynamic size locals)
  
  “std” `bp`

NOTE: recipes need to track `sp`, `bp`
HPCToolkit unwind recipe determination

Given an address A

1. Find start of procedure containing A (*)

2. use binary analysis of instruction semantics & some heuristics to determine location of RA, and location/value of bp (**)

3. Move to next instruction, repeat 2&3 till end of routine reached
Recipes are Intervals

For many instruction sequences, the recipe is the same for each instruction

So unwind recipes are stored as intervals
Some Notable features

• Algorithm is lazy
• No interprocedural analysis
• No intraprocedural control flow (straight line scan)
• We don't need 100% accuracy (but the heuristics seem quite good)
Recipe Heuristics

• No CFA => we need heuristics for multiple returns (epilogs)

• Use “canonical” frame instead
  Frequently the linearly closest with largest RA offset

• If “ret” is encountered with \( \text{loc}(\text{RA}) 
eq 0 \)
  then fix up intervals back to canonical
Libunwind (An Aside)

- Itanium version of HPCToolkit used libunwind for stackwalking
- Works great on itanium
- On x86, not so much
- We retain the interface: init_cursor, unw_step
Procedure Bounds

- x86 instructions are variable sized
- so recipe determination needs to start at the beginning of routines, and work forward
Procedure Bounds Determination

For each shared object (and the executable)

- Seed the list of candidates from the symbol information (thanks SymtabAPI)

- For each text segment, use binary analysis + heuristics to discover “hidden” functions
Hidden Procedure Bounds

• We assume that procedures are not split (*)
• No CFA is used. Linear scan for candidates
• ret or jmp makes next address a candidate
• backward branches may remove candidates
• push/pop pairs remove candidates
• various common ‘pad’ instructions ignored
Split procedures

• Some compilers actually split single procedures into multiple parts, and insert procedures between the parts !!

• Rather than pervert procedure bound determination, we introduce a special unwind heuristic
Split Unwind

IF:
• In routine R, last instruction is jmp T
• instruction just before T is jmp to begin(R)

THEN:
• Use recipe from T as basis for R, recompute all of R’s recipe intervals
How Effective Is It?

- For PFLOTRAN, the actual number was 148 out of 289M unwinds (8192 processors) on the Cray XT.

- SPEC benchmarks: compiled with PGI, Pathscale, Intel 292 out of 18M on Rice cluster.