Efficient Memory Tracing by Program Skeletonization

Alain Ketterlin Philippe Clauss

Université de Strasbourg (France)

INRIA (CAMUS team, Centre Nancy Grand-Est)

CNRS (LSIIT, UMR 7005)

2011 IEEE International Symposium
on Performance Analysis of Systems and Software (ISPASS)
April 10-12, 2011

Overview

The question is: How much of

1. the program, and
2. the input data

does one need to reproduce a full memory trace?

Larus' `qpt`:

- uses witnesses to reconstruct control-flow
- copies slices of instructions to a *trace generator*

The general idea is:

- use static analysis to reduce dynamic load

Overview

We target code like this (`312.swim_m`, `calc1`)

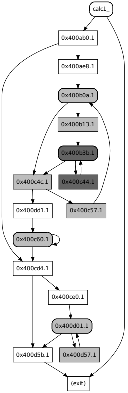
```
DO 100 J=1,N
  DO 100 I=1,M
    CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
    CV(I,J+1) = .5D0*(P(I,J+1)+P(I,J))*V(I,J+1)
    Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)-U(I+1,J)))
      /(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
    H(I,J) = P(I,J)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
        +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
100 CONTINUE
```

The goal of this work is:

- to be able to recognize such periodic behavior
- to minimize the "quantity" of instrumentation (statically = code bloat, dynamically = slowdown)
- to reproduce part of the work in the profiler

Symbolic Analysis

- Per routine
- Reconstruct the control-flow graph
  - indirect calls do not matter
  - indirect branches solved with heuristics
  - some functions remain un-analyzable
- Build a loop hierarchy
  - compute the dominator tree
  - duplicate bodies to solve irreducible loops
  - derive loop nesting
- Put the program in SSA form
  - all registers (`rax`, ..., `xmm0`, ..., `flags`)
  - except `rip`
  - memory as a unique variable `M`



Symbolic Analysis / Slicing

- SSA provides direct use-def links

```
...
0x400af5 mov eax, 0x603140      rax.8  ⇐

...
0x400b1d sub r13, 0xedb         r13.7  ⇐ r13.6
...
———                             rsi.9 = φ(rsi.8, rsi.10)
0x400b3b lea r11d, [rsi+0x1]    r11.6  ⇐ rsi.9
0x400b3f movsxd r10, r11d       r10.9  ⇐ r11.6
0x400b42 lea rdx, [r10+r13*1]   rdx.15 ⇐ r10.9, r13.7
...
0x400b4e lea r9, [rdx+0x...]    r9.9   ⇐ rdx.15
...
0x400b5c movsd xmm0, [rax+r9*8] xmm0.6 ⇐ M.22, rax.8, r9.9
                           └──→ 0xe28d4b0 + 8*rsi.9 + ....
```

- Substitution stops on:
  1. routine input parameters
  2. “non-linear” instructions
  3. memory accesses
  4. φ-nodes

Symbolic Analysis / Memory Addresses

- Compute a symbolic expression for each memory access
- Hope that many addresses are based on few definitions

```
movsd xmm0, q[rax+r9*8]
                  └──→ 0xe28d4b0 + 8*rsi.9 + 30416*r15.6
addsd xmm0, q[rax+rbx*8]
                  └──→ 0xe28d4a8 + 8*rsi.9 + 30416*r15.6
mulsd xmm0, xmm4
mulsd xmm0, q[rax+rdx*8]
                  └──→  0x5fba70 + 8*rsi.9 + 30416*r15.6
movsd q[rax+rdx*8+0x...], xmm0
               └──────→0x3e68b090 + 8*rsi.9 + 30416*r15.6
[...]
```

- The real code has 20+ accesses, based on 3 registers

Symbolic Analysis / Induction Variable Resolution

- Loops define another level of repetitive behavior
- Induction variables are definitions whose values depend only on the (normalized) iteration number
- They appear as φ-nodes on loop heads

```
0x400b36   mov esi, 0x1               rsi.8  ⇐            = 0x1
           rsi.9 = φ(rsi.8, rsi.10)   = (0x1) + J*(0x1)
0x400b3b   lea r11d, [rsi+0x1]        r11.6  ⇐ rsi.9      = 0x1+rsi.9
...
0x400c44   mov esi, r11d              rsi.10 ⇐ r11.6      = 0x1+rsi.9
0x400c47   jmp 0x400b3b
```

- IV resolution: on loop heads

$$\text{if} \quad r = \varphi(\alpha, r + \beta) \quad \text{then} \quad r = \alpha + I \times \beta$$

iff α and β are loop-invariant; I is a normalized counter

Symbolic Analysis / Induction Variable Resolution

- Our previous example:

```
movsd xmm0, q[rax+r9*8]
                 └────→ 0xe28d4b0 + 8*rsi.9 + 30416*r15.6
                    =   0xe294b88 + 8*J + 30416*I
addsd xmm0, q[rax+rbx*8]
                 └────→ 0xe28d4a8 + 8*rsi.9 + 30416*r15.6
                    =   0xe294b80 + 8*J + 30416*I
mulsd xmm0, xmm4
mulsd xmm0, q[rax+rdx*8]
                 └────→  0x5fba70 + 8*rsi.9 + 30416*r15.6
                    =    0x603148 + 8*J + 30416*I
movsd q[rax+rdx*8+0x...], xmm0
               └──────→0x3e68b090 + 8*rsi.9 + 30416*r15.6
                    = 0x3e692768 + 8*J + 30416*I
[...]
```

- The real code: 20+ accesses, only 1 register left

Symbolic Analysis / Branch Conditions

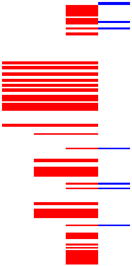
- Capturing the control-flow: obtain symbolic conditions
  1. the branch provides the comparison
  2. the definition of `rflags` provides the expression
- Linear expressions compared to zero with $<, \leq, >, \geq, =, \neq$
- Example:

```
0x400b0a            ...

         0x400b3b   ...
         ...
         0x400c3f   cmp esi, r14d
         0x400c42   jz 0x400c4c           ?  0x1+J-r14.5 == 0
         ...
         0x400c47   jmp 0x400b3b

0x400c4c            cmp r15d, d[rsp-0x4]
0x400c51            jz 0x400dd1           ?  (unknown)
...
0x400c5b            jmp 0x400b0a
0x400dd1
```

- Unknown conditions need instrumentation

Symbolic Analysis / Results

- Implemented with Pin
- Memory accesses vs. register definitions

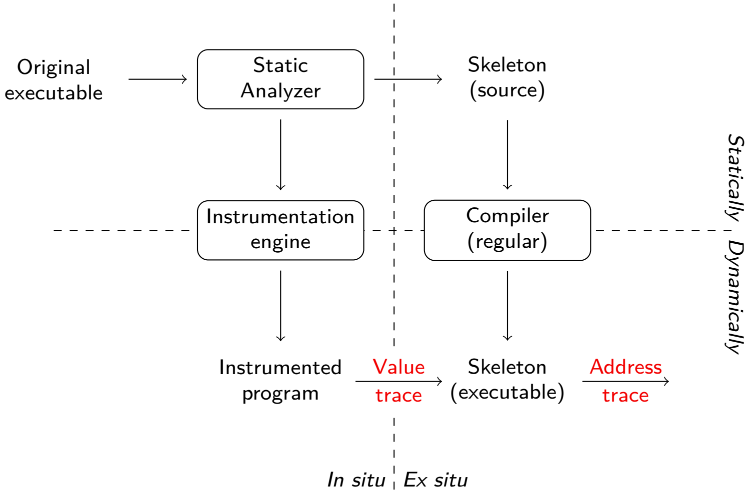


Program	Static Ratio	Dynamic Ratio
310.wupwise_m	0.241	0.261
312.swim_m	0.152	6e-4
429.mcf	0.413	0.892
average	0.26	0.24

Memory Tracing

- Naive approach: instrument every memory access
- However, this incurs:
  - code bloat
  - massive slowdowns
- Program skeletonization is:
  - instrument only the required (register) definitions
  - let the profiler compute effective addresses

Memory Tracing / Architecture



Memory Tracing / The Skeleton

The skeleton...

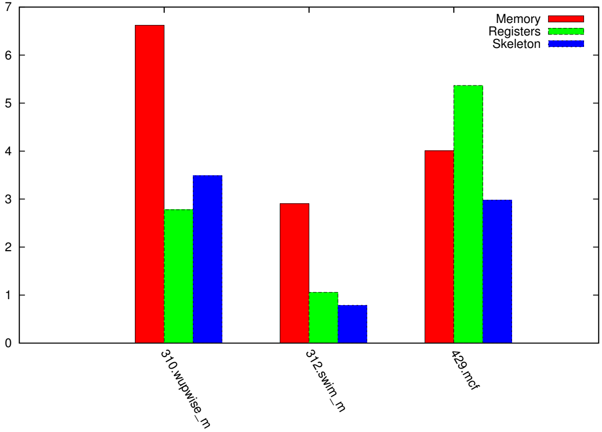
- is built directly from the CFG
  - actually, from the loop hierarchy
- inputs raw register definition values
- contains expressions for
  - memory addresses
  - (some) branch conditions
- maintains loop counters
- is generated as C code
- has the same structure as the program (sampling, partial tracing...)

Memory Tracing / The Skeleton

```
B_0x400ae8:
  ...
  reg_t r14_5 = IN();
L_0x400b0a:
  reg_t I = 0;
B_0x400b0a:
  if ( r14_5 <= 0 ) goto B_0x400c4c;
B_0x400b13:  /* empty, not generated */

    L_0x400b3b:
      reg_t J = 0;
    B_0x400b3b:
      OUT(0x400b5c,'R',8, 0xe294b88 + 8*J + 30416*I );
      OUT(0x400b62,'R',8, 0xe294b80 + 8*J + 30416*I );
      OUT(0x400b6b,'R',8, 0x603148 + 8*J + 30416*I );
      OUT(0x400b70,'W',8, 0x3e692768 + 8*J + 30416*I );
      ...
      if ( 1 + J - r14_5 == 0 ) goto B_0x400c4c;
    B_0x400c44:
      J = J + 1;
      goto B_0x400b3b;

B_0x400c4c:
  if ( IN() ) goto B_0x400dd1;
B_0x400c57:
  I = I + 1;
  goto B_0x400b0a;
```

Memory Tracing / Results

- Running times (normalized)



- $\max(\text{Values}, \text{Skeleton})/\text{Memory} = 0.61$

Conclusion

- The skeleton
  - is a compressed form of the original program
  - is portable and independent of the original program
- The input trace
  - provides un-reproducible data
  - contains just enough data
- Reproducing the trace may be done
  - on-line, with the skeleton running in parallel with the program
  - off-line, by running the skeleton off a stored trace
- Obtaining more speed/compression requires
  - more powerful static analysis