Recovering the Memory Behavior of Executable Programs

Alain KETTERLIN   Philippe CLAUSS

Université de Strasbourg (France)

INRIA CAMUS team

SCAM 2010, 12th–13th September
Timişoara (Romania)

Motivation

Research topics

- How does that program access memory?
- Decompiling x86-64 executables
  - We use libraries, and multi-language applications
  - We need some dynamic capabilities
- Dynamic analysis/optimization/parallelization

Client applications

- Prediction for local adjustments (e.g., prefetching)
- Optimize memory tracing (e.g., for cache simulation)
- Evaluate memory consumption
- Detect parallel sections (and parallelize them)

(Anything related to high-performance computing.)

Loop Nests, Access Functions, and Polyhedra

The target model

- loops, accessing arrays (or memory):
  - with linear access functions $f_n, f_m, \ldots$
  - on loop counters $i_0, i_1, \ldots$
  - and parameters $\vec{r}$

```
for i0 = 0 to u0(r)
    . . .
        for ip = 0 to up(i0,...,ip-1,r)
            ⋮
            M[f8(i0,...,ip,r)]
                = g3 (M[f6(i0,...,ip,r)],M[f7(...)],...)
            ⋮
```

$$
\begin{aligned}
&\texttt{for } i_0 = 0 \texttt{ to } u_0(\vec{r})\\
&\quad \ddots\\
&\qquad \texttt{for } i_p = 0 \texttt{ to } u_p(i_0,\ldots,i_{p-1},\vec{r})\\
&\qquad\quad \vdots\\
&\qquad\quad \texttt{M[}f_8(i_0,\ldots,i_p,\vec{r})\texttt{]}\\
&\qquad\qquad = g_3\,(\texttt{M[}f_6(i_0,\ldots,i_p,\vec{r})\texttt{]},\texttt{M[}f_7(\ldots)\texttt{]},\ldots)\\
&\qquad\quad \vdots
\end{aligned}
$$

- We know how to deal with such loop nests, i.e., with polyhedral (linear programming) techniques

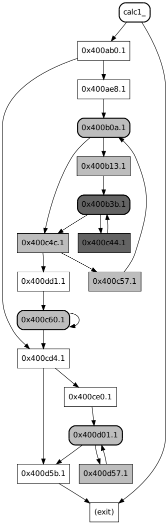
Preliminary Analysis

Parsing binaries

- With Pin (for parsing and instrumentation)
- Per-routine analysis
- Rebuild control-flow graph
- Compute dominator tree
- Extract a loop hierarchy

Traps

- Some indirect branches
  ($\rightarrow$ compiler-specific heuristics)
- Some irreducible loops
  ($\rightarrow$ technical complications)



Static Single Assignment

Variables

- GP registers: `rsp`, `rax`, `rbx`, ...
- SSE registers: `xmm1`, `xmm2`, ...
- Flag register: `rflags`
- Memory: `M`, as a whole, with weak updates

Instructions

- use some registers (i.e., versions) and define *new* registers
- for instance: `sub rax, 0x10` really means:

  `(rax.7,rflags.19) ⇐ SUB(rax.6,0x10)`

- memory stores use the previous version of memory
  for instance: `mov [rsp-0x4], rax` really means:

  `M.9 ⇐ MOV(M.8,rax.3,rsp.8)`

Symbolic Expansion

Follow 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
```

Example

Expansion stops at

- initial values (routine entry block)
- φ-functions (but see below)
- memory accesses (internal data-flow only)
- any instruction beyond the linear integer model

Example

```
movsd xmm0,[rax+r9*8]  @ 0xe28d4b0+8*rsi.9+30416*r15.6
addsd xmm0,[rax+rbx*8] @ 0xe28d4a8+8*rsi.9+30416*r15.6
...
mulsd xmm0,[rax+rdx*8] @  0x5fba70+8*rsi.9+30416*r15.6
...
movsd xmm0,[rax+r8*8]  @ 0xe294b78+8*rsi.9+30416*r15.6
addsd xmm0,[rax+rbx*8] @ 0xe28d4a8+8*rsi.9+30416*r15.6
...
```

Induction Variable Resolution

Induction variables

- φ-functions on loop-heads
- expand external variants $\rightarrow$ initial value α
- expand internal variants $\rightarrow$ recurrence
- match recurrence with $\varphi + \beta$
- introduce a normalized counter: $\alpha + \mathtt{I} \cdot \beta$

```
r.0 = α
r.1 = ϕ(r.0,r.2)
r.2 = r.1 + β
```

Example

```
mov r15d, 0x1          r15.5 ⇐ 0x1
                       r15.6 = ϕ(r15.7, r15.5)     (0x1) + I*(0x1)

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

add r15d, 0x1          r15.7, rf.29 ⇐ r15.6        = 0x1+r15.6
```

Loops

After register expansion

```
for I = 0 to ?
    for J = 0 to ?
        ...
        @ 0xe294b88 + 8*J + 30416*I
        @ 0xe294b80 + 8*J + 30416*I
        @  0x603148 + 8*J + 30416*I
        ...
        @ 0xe29c250 + 8*J + 30416*I
        @ 0xe294b80 + 8*J + 30416*I
```

Recovering the bounds

- Derive symbolic branching conditions
- Control-flow analysis combines conditions, define trip counts

Evaluation (static)

Methodology

- SPEC programs
- compiled with regular gcc
- measure:

$$\text{instrumentation ratio} = \frac{\text{number of registers needed}}{\text{number of memory accesses}}$$

- results:

Suite	**Progs**	**-O1**	**-O3**
SPEC OMP-2001	11	0.19	0.16
SPEC CPU-2006 (FP)	17	0.22	0.20
SPEC CPU-2006 (INT)	11	0.32	0.30

(a total of 39 programs)

Tracing

Problem statement

- all memory accesses of the program, for, e.g.,
  - cache simulation
  - dependence analysis
  - ...
- incurs massive slowdowns ($\approx \frac{1}{3}$ accesses per inst.)
  - but completely accurate

Principle

- Use symbolic expansion to extract:
  - memory addresses
  - branch conditions
- Instrument register definitions instead of memory accesses
- Let the profiler track the execution and compute effective addresses

Evaluation (dynamic)

Results

- same program set, two data sets
- measure:

$$\text{dynamic instrumentation ratio} = \frac{\text{number of registers set}}{\text{number of memory accesses}}$$

- results:

Data	Suite	`-O1`	`-O3`
test	SPEC OMP-2001	0.16	0.12
	SPEC CPU-2006 (FP)	0.28	0.24
	SPEC CPU-2006 (INT)	0.78	0.77
ref	SPEC OMP-2001	0.16	0.13
	SPEC CPU-2006 (FP)	0.24	0.19
	SPEC CPU-2006 (INT)	0.78	0.79

Conclusion

On Tracing

- Surprisingly effective
- Divides tracing time by 2 to 3000 on data-intensive progs
- Pathological cases are easy to detect/handle

On Extracting Usable Models

- Almost perfect on “leaf” loop
  ($\rightarrow$ vectorization)
- Limited by stack access/spills/...
  ($\rightarrow$ requires primitive points-to analysis to resolve locals)

Controversial statement(s)

- Source code is *not* the right level for parallelism detection, parallelization, . . . , memory consumption, . . .
  (anything related to performance re. memory)
- Leave more work for run time
  (the compiler should provide alternatives/parameters only)