Recovering the Memory Behavior of Executable Programs

Alain KETTERLIN    Philippe CLAUSS

Université de Strasbourg (France)
INRIA CAMUS team

SCAM 2010, 12\textsuperscript{th}–13\textsuperscript{th} September
Timișoara (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.)
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 $\vec{r}$

```plaintext
for $i_0 = 0$ to $u_0(\vec{r})$
  
  ...$

  for $i_p = 0$ to $u_p(i_0, \ldots, i_{p-1}, \vec{r})$
    
    ...$

    $M[f_8(i_0, \ldots, i_p, \vec{r})]$
    $= g_3(M[f_6(i_0, \ldots, i_p, \vec{r})], M[f_7(\ldots)], \ldots)$
    
    ...
```

- 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 (→ compiler-specific heuristics)
- Some irreducible loops (→ technical complications)
Variables

- GP registers: \( \text{rsp}, \text{rax}, \text{rbx}, \ldots \)
- SSE registers: \( \text{xmm1}, \text{xmm2}, \ldots \)
- Flag register: \( \text{rflags} \)
- Memory: \( M \), as a whole, with weak updates

Instructions

- use some registers (i.e., versions) and define new registers
- for instance: \( \text{sub rax, 0x10} \) really means:
  \[
  (\text{rax.7}, \text{rflags.19}) \leftarrow \text{SUB(\text{rax.6, 0x10})}
  \]
- memory stores use the previous version of memory
  for instance: \( \text{mov [rsp-0x4], rax} \) really means:
  \[
  M.9 \leftarrow \text{MOV(M.8, rax.3, rsp.8)}
  \]
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 r1ld, [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
Expansion stops at

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

Example

```plaintext
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

- \( \varphi \)-functions on loop-heads
- expand external variants \( \rightarrow \) initial value \( \alpha \)
- expand internal variants \( \rightarrow \) recurrence
- match recurrence with \( \varphi + \beta \)
- introduce a normalized counter: \( \alpha + I \cdot \beta \)

Example

\[
\begin{align*}
\text{mov } r15d, \ 0x1 & \quad r15.5 \leftarrow 0x1 \\
& \quad r15.6 = \varphi(r15.7, r15.5) \quad (0x1) + I \cdot (0x1) \\
\text{mov } esi, \ 0x1 & \quad rs1.8 \leftarrow 0x1 \\
& \quad rs1.9 = \varphi(rs1.8, rs1.10) \quad (0x1) + J \cdot (0x1) \\
\text{lea } r11d, \ [rsi+0x1] & \quad r11.6 \leftarrow rs1.9 \\
& \quad = 0x1 + rs1.9 \\
\ldots \end{align*}
\]

\[
\begin{align*}
\text{mov } esi, \ r11d & \quad rs1.10 \leftarrow r11.6 \\
& \quad = 0x1 + rs1.9 \\
\text{add } r15d, \ 0x1 & \quad r15.7, \ rf.29 \leftarrow r15.6 \\
& \quad = 0x1 + r15.6
\end{align*}
\]
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:

<table>
<thead>
<tr>
<th>Suite</th>
<th>Progs</th>
<th>-O1</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC OMP-2001</td>
<td>11</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>SPEC CPU-2006 (FP)</td>
<td>17</td>
<td>0.22</td>
<td>0.20</td>
</tr>
<tr>
<td>SPEC CPU-2006 (INT)</td>
<td>11</td>
<td>0.32</td>
<td>0.30</td>
</tr>
</tbody>
</table>

(a total of 39 programs)
Problems 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:

<table>
<thead>
<tr>
<th>Data</th>
<th>Suite</th>
<th>-O1</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>SPEC OMP-2001</td>
<td>0.16</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>SPEC CPU-2006 (FP)</td>
<td>0.28</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>SPEC CPU-2006 (INT)</td>
<td>0.78</td>
<td>0.77</td>
</tr>
<tr>
<td>ref</td>
<td>SPEC OMP-2001</td>
<td>0.16</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>SPEC CPU-2006 (FP)</td>
<td>0.24</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>SPEC CPU-2006 (INT)</td>
<td>0.78</td>
<td>0.79</td>
</tr>
</tbody>
</table>
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
  (→ vectorization)
- Limited by stack access/spills/...
  (→ 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)