Pin: building customized program analysis tools with dynamic instrumentation

C.-K. Luk & a bunch of other Intel guys

presented by Bjoern Doebel
## Program analysis

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC, (SP)Lint, Coverity, ...</td>
<td>Valgrind, Pin, DynamoRIO, ...</td>
</tr>
<tr>
<td>- Compile-time</td>
<td>- Runtime</td>
</tr>
<tr>
<td>- Heavy-weight</td>
<td>- Trade-off overhead vs. realistic observations</td>
</tr>
<tr>
<td>- No environmental information</td>
<td>- Path coverage issues</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source-level</th>
<th>Binary analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lint, Coverity</td>
<td>Bitblaze/Vine, Valgrind, Pin, ...</td>
</tr>
<tr>
<td>- Exact information</td>
<td>- Inexact w.r.t. source code</td>
</tr>
<tr>
<td>- Problem: 3rd party tools</td>
<td>- Support for any kind of application / library</td>
</tr>
</tbody>
</table>
Dynamic binary analysis

• Approach:
  – Disassemble binary (→ intermediate language)
  – Insert tool-specific instrumentation at IL level
  – Recompile into machine code

• Usually JIT-based
  – Disassemble & Resynthesize
  – Copy & Annotate

• Main focus of research: runtime optimizations
Figure 2. Pin’s software architecture
• No intermediate language
• Trace-based recompilation
• Can attach to running program
Optimization 1: Trace linking

0x40000000
  jmp [%eax]

0x70000000
  mov [%eax], %edx
  jmp $0x70001000

0x70001000
  lea -0x40001000(%edx), %ecx
  jecxz $match1
  jmp $0x70002000

match1:
  ...

0x70002000
  lea -0x40002000(%edx), %ecx
  jecxz $match2
  jmp $LookupHTab_1

match2:
  ...

LookupHTab_1
  mov %edx, %esi
  and $0x3ff, %esi
  cmp 0x30898200(,%esi,8), %edx
  jnz $VMEntry # miss
  jmp 0x30898204(,%esi,8) # hit

VM
(b) Using cloning to help predict return targets

Figure 3. Compiling indirect jumps and returns
Optimization 3: Register Re-allocation

(b) Valgrind’s approach

Trace 1

mov $1, %eax
mov $2, %ebx
cmp %ecx, %edx
jz t
...
add $1, %eax
sub $2, %ebx
...
mov $1, %eax
mov $2, %esi
cmp %ecx, %edx
mov %eax, EAX
mov %esi, EBX
jz t'

Trace 2

t':

mov EAX, %eax
mov EBX, %edi
add $1, %eax
sub $2, %edi
...

Optimization 3: Register Re-allocation

(c) Pin (no reconciliation needed)

Compile Trace 2 using the bindings:

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%eax</td>
</tr>
<tr>
<td>%ebx</td>
<td>%esi</td>
</tr>
<tr>
<td>%ecx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%edx</td>
<td>%edx</td>
</tr>
</tbody>
</table>

Trace 1
mov $1, %eax
mov $2, %esi
cmp %ecx, %edx
jz t
...

Trace 2
add $1, %eax
sub $2, %esi
...

t:
mov $1, %eax
mov $2, %ebx
cmp %ecx, %edx
jz t
...

t':
add $1, %eax
sub $2, %esi
...
Optimization 3: Register Re-allocation

(d) Pin (minimal reconciliation needed)

Trace 1 (being compiled)

```
mov $1, %eax
mov $2, %ebx
cmp %ecx, %edx
jz t
...
add $1, %eax
sub $2, %ebx
...
```

No need to recompile Trace 2, simply reconcile the bindings of virtual %ebx in Traces 1 and 2.

Trace 2 (previously compiled)

```
t':
add $1, %eax
sub $2, %edi
...
```
Performance evaluation

(b) With basic-block counting

<table>
<thead>
<tr>
<th></th>
<th>Valgrind</th>
<th>DynamoRIO</th>
<th>Pin/IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>682</td>
<td>786</td>
<td>236</td>
</tr>
<tr>
<td>crafty</td>
<td>1091</td>
<td>1170</td>
<td>838</td>
</tr>
<tr>
<td>gap</td>
<td>343</td>
<td>324</td>
<td>259</td>
</tr>
<tr>
<td>gcc</td>
<td>860</td>
<td>851</td>
<td>606</td>
</tr>
<tr>
<td>gzip</td>
<td>1533</td>
<td>1431</td>
<td>1182</td>
</tr>
<tr>
<td>mcf</td>
<td>189</td>
<td>211</td>
<td>171</td>
</tr>
<tr>
<td>parser</td>
<td>194</td>
<td>206</td>
<td>108</td>
</tr>
<tr>
<td>perl2mk</td>
<td>793</td>
<td>794</td>
<td>179</td>
</tr>
<tr>
<td>twolf</td>
<td>574</td>
<td>575</td>
<td>412</td>
</tr>
<tr>
<td>vortex</td>
<td>934</td>
<td>934</td>
<td>408</td>
</tr>
<tr>
<td>vpr</td>
<td>936</td>
<td>936</td>
<td>320</td>
</tr>
<tr>
<td>NT-AriMean</td>
<td>834</td>
<td>834</td>
<td>251</td>
</tr>
</tbody>
</table>
• Is there any advantage from having no intermediate language?
  – In fact, the compiler has some kind of intermediate representation, even if it is no language.

• Comparing Pin with Valgrind: How would it perform, if Pin were required to provide
  – Shadow values
  – Address space management
  – System call interception
• “[...] We implemented basic-block counting by modifying a tool in the Valgrind package named lackey [...]” [1]

  “Lackey is a simple Valgrind tool that does various kinds of basic program measurement. It adds quite a lot of simple instrumentation to the program's code. It is primarily intended to be of use as an example tool, and consequently emphasises clarity of implementation over performance.” [2]

[1] The paper