Dynamic Binary Translation & Instrumentation

Pin
Building Customized Program Analysis Tools with Dynamic Instrumentation

CK Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Kim Hazelwood
Intel

Vijay Janapa Reddi
University of Colorado

http://rogue.colorado.edu/Pin
Instrumentation

- Insert extra code into programs to collect information about execution
  - Program analysis:
    - Code coverage, call-graph generation, memory-leak detection
  - Architectural study:
    - Processor simulation, fault injection

- Existing binary-level instrumentation systems:
  - Static:
    - ATOM, EEL, Etch, Morph
  - Dynamic:
    - Dyninst, Vulcan, DTrace, Valgrind, Strata, DynamoRIO

Pin is a new dynamic binary instrumentation system

A Pintool for Tracing Memory Writes

```c
#include <iostream>
#include "pin.H"

FILE* trace;

VOID RecordMemWrite(VOID* ip, VOID* addr, UINT32 size) {
  fprintf(trace, "%p: W %p %d\n", ip, addr, size);
}

VOID Instruction(INS ins, VOID* v) {
  if (INS_IsMemoryWrite(ins))
    INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite), IARG_INST_PTR,
                   IARG_MEMORYWRITE_EA, IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char* argv[]) {
  PIN_Init(argc, argv);
  trace = fopen("atrace.out", "w");
  INS_AddInstrumentFunction(Instruction, 0);
  PIN_StartProgram();
  return 0;
}
```

- Same source code works on the 4 architectures
  => Pin takes care of different addressing modes
- No need to manually save/restore application state
  => Pin does it for you automatically and efficiently
Dynamic Instrumentation

Original code

Code cache

Pin fetches trace starting block 1 and start instrumentation

 Pins transfers control into code cache (block 1)
Dynamic Instrumentation

Original code

Code cache

1. Pin fetches and instrument a new trace

Pin’s Software Architecture

Address space

- 3 programs (Pin, Pintool, App) in same address space:
  - User-level only
- Instrumentation APIs: Through which Pintool communicates with Pin
- JIT compiler: Dynamically compile and instrument
- Emulation unit: Handle instrs that can't be directly executed (e.g., sycalls)
- Code cache: Store compiled code
- Coordinated by VM

3 programs (Pin, Pintool, App) in same address space:
- User-level only
- Instrumentation APIs: Through which Pintool communicates with Pin
- JIT compiler: Dynamically compile and instrument
- Emulation unit: Handle instrs that can't be directly executed (e.g., sycalls)
- Code cache: Store compiled code
- Coordinated by VM
Pin Internal Details

- Loading of Pin, Pintool, & Application
- An Improved Trace Linking Technique
- Register Re-allocation
- Instrumentation Optimizations
- Multithreading Support

Register Re-allocation

- Instrumented code needs extra registers. E.g.:
  - Virtual registers available to the tool
  - A virtual stack pointer pointing to the instrumentation stack
  - Many more...

- Approaches to get extra registers:
  1. Ad-hoc (e.g., DynamoRIO, Strata, DynInst)
     - Whenever you need a register, spill one and fill it afterward
  2. Re-allocate all registers during compilation
     a. Local allocation (e.g., Valgrind)
        • Allocate registers independently within each trace
     b. Global allocation (Pin)
        • Allocate registers across traces (can be inter-procedural)
**Valgrind’s Register Re-allocation**

**Original Code**

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

```
t:
```

**Trace 1**

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
mov %eax, % edi
mov %esi, %edi
jt t'
```

```
re-allocate
```

```
Trace 2
```

```
mov %eax, %edi
mov %esi, %edi
add 1, %eax
sub 2, %edi
```

---

**Simple but inefficient**

- All modified registers are spilled at a trace’s end
- Refill registers at a trace’s beginning

---

**Pin’s Register Re-allocation**

**Scenario (1): Compiling a new trace at a trace exit**

**Original Code**

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

```
t:
```

**Trace 1**

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
jt t'
```

```
re-allocate
```

```
Trace 2
```

```
add 1, %eax
sub 2, %esi
... 
```

---

**No spilling/filling needed across traces**
Pin’s Register Re-allocation

Scenario (2): Targeting an already generated trace at a trace exit

**Original Code**

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

**Trace 1 (being compiled)**

```
mov 1, %eax
mov 2, %esi
cmp %ecx, %edx
mov %esi, SPILL ebx
mov SPILL ebx, %edi
jz t'
add 1, %eax
sub 2, %edi
```

**Trace 2 (in code cache)**

```
t':
... 
```

**Minimal spilling/filling code**

Instrumentation Optimizations

1. Inline instrumentation code into the application

2. Avoid saving/restoring eflags with liveness analysis

3. Schedule inlined instrumentation code
**Example: Instruction Counting**

**Original code**
```
cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>
add %ecx, %edx
cmp %edx, 0
je <target2>
```

**Instrument without applying any optimization**
```
bBL_InsertCall(bbl, IPOINT_BEFORE, docount(), IARG_UINT32, BBL_NumIns(bbl), IARG_END)
```

**Trace**
```
mov %esp, SPILLappsp
mov SPILLpinsp, %esp
call <bridge>
cmov %esi, %edi
mov SPILLappsp, %esp
call <bridge>
add %ecx, %edx
cmp %edx, 0
je <target2'>
```

**33 extra instructions executed altogether**

**Inlining**
```
cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>
add %ecx, %edx
cmp %edx, 0
je <target2>
```

**Trace**
```
mov %esp, SPILLappsp
mov SPILLpinsp, %esp
pushf
add 0x3, %eax
push %ecx
push %edx
movl 0x3, %eax
call docount
pop %eax
pop %ecx
pop %edx
popf
ret
```

**11 extra instructions executed**
Example: Instruction Counting

Original code

```
cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>
add %ecx, %edx
cmp %edx, 0
je <target2>
```

Inlining + eflags liveness analysis

```
move %esp, SPILLappsp
mov SPILLpinsp, %esp
pushf
add 0x3, icount
popf
cmov %esi, %edi
mov SPILLappsp, %esp
cmp %edi, (%esp)
jle <target1'>
```

7 extra instructions executed

```
add 0x3, icount
add %ecx, %edx
cmp %edx, 0
je <target2'>
```

Example: Instruction Counting

Original code

```
cmov %esi, %edi
cmp %edi, (%esp)
jle <target1>
add %ecx, %edx
cmp %edx, 0
je <target2>
```

Inlining + eflags liveness analysis + scheduling

```
cmov %esi, %edi
add 0x3, icount
cmp %edi, (%esp)
jle <target1'>
```

2 extra instructions executed

```
add 0x3, icount
add %ecx, %edx
cmp %edx, 0
je <target2'>
```
Pin Instrumentation Performance

Runtime overhead of basic-block counting with Pin on IA32

- Without optimization
- Inlining
- Inlining + eflags liveness analysis
- Inlining + eflags liveness analysis + scheduling

(SPEC2K using reference data sets)

Comparison among Dynamic Instrumentation Tools

Runtime overhead of basic-block counting with three different tools

- Valgrind is a popular instrumentation tool on Linux
- Call-based instrumentation, no inlining
- DynamoRIO is the performance leader in binary dynamic optimization
- Manually inline, no eflags liveness analysis and scheduling

Pin automatically provides efficient instrumentation
Pin Applications

• Sample tools in the Pin distribution:
  • Cache simulators, branch predictors, address tracer, syscall tracer, edge profiler, stride profiler

• Some tools developed and used inside Intel:
  • Opcodemix (analyze code generated by compilers)
  • PinPoints (find representative regions in programs to simulate)
  • A tool for detecting memory bugs

• Some companies are writing their own Pintools:
  • A major database vendor, a major search engine provider

• Some universities using Pin in teaching and research:
  • U. of Colorado, MIT, Harvard, Princeton, U of Minnesota, Northeastern, Tufts, University of Rochester, ...

Conclusions

• Pin
  • A dynamic instrumentation system for building your own program analysis tools
  • Easy to use, robust, transparent, efficient
  • Tool source compatible on IA32, EM64T, Itanium, ARM
  • Works on large applications
    • database, search engine, web browsers, ...
  • Available on Linux; Windows version coming soon

• Downloadable from http://rogue.colorado.edu/Pin
  • User manual, many example tools, tutorials
  • 3300 downloads since 2004 July
Valgrind
A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote — National ICT Australia
Julian Seward — OpenWorks LLP

FAQ #1

• How do you pronounce “Valgrind”?

• “Val-grinned”, not “Val-grined”

• Don’t feel bad: almost everyone gets it wrong at first
DBA tools

- Program analysis tools are useful
  - Bug detectors
  - Profilers
  - Visualizers

- **Dynamic binary analysis** (DBA) tools
  - Analyse a program’s machine code at run-time
  - Augment original code with **analysis code**

Building DBA tools

- **Dynamic binary instrumentation** (DBI)
  - Add analysis code to the original machine code at run-time
  - No preparation, 100% coverage

- DBI frameworks
  - Pin, DynamoRIO, Valgrind, etc.
Prior work

<table>
<thead>
<tr>
<th>Well-studied</th>
<th>Not well-studied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework performance</td>
<td>Instrumentation capabilities</td>
</tr>
<tr>
<td>Simple tools</td>
<td>Complex tools</td>
</tr>
</tbody>
</table>

- Potential of DBI has not been fully exploited
  - Tools get less attention than frameworks
  - Complex tools are more interesting than simple tools

Shadow value tools
# Shadow value tools (I)

- Shadow every value with another value that describes it
- Tool stores and propagates shadow values in parallel

<table>
<thead>
<tr>
<th>Tool(s)</th>
<th>Shadow values help find...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcheck</td>
<td>Uses of undefined values</td>
</tr>
<tr>
<td>Annelid</td>
<td>Array bounds violations</td>
</tr>
<tr>
<td>Hobbes</td>
<td>Run-time type errors</td>
</tr>
<tr>
<td>TaintCheck, LIFT, TaintTrace</td>
<td>Uses of untrusted values</td>
</tr>
<tr>
<td>“Secret tracker”</td>
<td>Leaked secrets</td>
</tr>
<tr>
<td>DynCompB</td>
<td>Invariants</td>
</tr>
</tbody>
</table>

## Memcheck

- Shadow values: defined or undefined

<table>
<thead>
<tr>
<th>Original operation</th>
<th>Shadow operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int* p = malloc(4)</td>
<td>sh(p) = undefined</td>
</tr>
<tr>
<td>R1 = 0x12345678</td>
<td>sh(R1) = defined</td>
</tr>
<tr>
<td>R1 = R2</td>
<td>sh(R1) = sh(R2)</td>
</tr>
<tr>
<td>R1 = R2 + R3</td>
<td>sh(R1) = addsh(R2, R3)</td>
</tr>
<tr>
<td>if R1==0 then goto L</td>
<td>complain if sh(R1) is undefined</td>
</tr>
</tbody>
</table>

- 30 undefined value bugs found in OpenOffice
Shadow value tools (II)

- All shadow value tools work in the same basic way

- Shadow value tools are **heavyweight** tools
  - Tool’s data + ops are as complex as the original program’s

- Shadow value tools are hard to implement
  - Multiplex real and shadow registers onto register file
  - Squeeze real and shadow memory into address space
  - Instrument most instructions and system calls

Valgrind basics
Valgrind

- Software
  - Free software (GPL)
  - {x86, x86-64, PPC}/Linux, PPC/AIX

- Users
  - Development: Firefox, OpenOffice, KDE, GNOME, MySQL, Perl, Python, PHP, Samba, RenderMan, Unreal Tournament, NASA, CERN
  - Research: Cambridge, MIT, Berkeley, CMU, Cornell, UNM, ANU, Melbourne, TU Muenchen, TU Graz

- Design
  - Heavyweight tools are well supported
  - Lightweight tools are slow

Two unusual features of Valgrind
#1: Code representation

**D&R**
Disassemble-and-resynthesize (Valgrind)

- Disassemble
- Instrument
- Resynthesize

**C&A**
Copy-and-annotate

- Annotate
- Instrument
- Interleave

Pros and cons of D&R

- **Cons:** Lightweight tools
  - Framework design and implementation effort
  - Code translation cost, code quality

- **Pros:** Heavyweight tools
  - Analysis code as expressive as original code
  - Tight interleaving of original code and analysis code
  - Obvious when things go wrong!
Other IR features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-class shadow</td>
<td>As expressive as normal</td>
</tr>
<tr>
<td>registers</td>
<td>registers</td>
</tr>
<tr>
<td>Typed, SSA</td>
<td>Catches instrumentation</td>
</tr>
<tr>
<td>RISC-like</td>
<td>errors</td>
</tr>
<tr>
<td>Infinitely many</td>
<td>Fewer cases to handle</td>
</tr>
<tr>
<td>temporaries</td>
<td>Never have to find a spare register</td>
</tr>
</tbody>
</table>

• Writing complex inline analysis code is easy

#2: Thread serialisation

• Shadow memory: memory accesses no longer atomic
  • Uni-processors: thread switches may intervene
  • Multi-processors: real/shadow accesses may be reordered

• Simple solution: serialise thread execution!
  • Tools can ignore the issue
  • Great for uni-processors, slow for multi-processors...
## Performance

### SPEC2000 Performance

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valgrind, no-instrumentation</td>
<td>4.3x</td>
</tr>
<tr>
<td>Pin/DynRIO, no-instrumentation</td>
<td>~1.5x</td>
</tr>
<tr>
<td>Memcheck</td>
<td>22.1x (7--58x)</td>
</tr>
<tr>
<td>Most other shadow value tools</td>
<td>10--180x</td>
</tr>
<tr>
<td>LIFT</td>
<td>3.6x (*)</td>
</tr>
</tbody>
</table>

(*) LIFT limitations:
- No FP or SIMD programs
- No multi-threaded programs
- 32-bit x86 code on 64-bit x86 machines only
Post-performance

- Only Valgrind allows robust shadow value tools
  - All robust ones built with Valgrind or from scratch

- Perception: “Valgrind is slow”
  - Too simplistic
  - Beware apples-to-oranges comparisons
  - Different frameworks have different strengths

Future of DBI
The future

• Interesting tools!
  • Memcheck changed many C/C++ programmer’s lives
  • Tools don’t arise in a vacuum

• What do you want to know about program execution?
  • Think big!
  • Don’t worry about being practical at first

If you remember nothing else...
Take-home messages

- Heavyweight tools are interesting
- Each DBI framework has its pros and cons
- Valgrind supports heavy-weight tools well

www.valgrind.org