Dynamic Instrumentation with Pin

Robert Cohn
Intel
Pin

- Fine-grained dynamic instrumentation of user mode programs
- Instrumentation
  - Inserting extra code to observe/change program
  - Profilers, trace collectors, ...
- Dynamic
  - Done at run-time, no special compilation or linking
  - Adapt instrumentation during execution
- Fine-grained
  - Observe the execution of every instruction
  - Request instrumentation before or after any instruction execution
- Transparent
  - Instrumentation observes original program
#include <stdio.h>
#include "pin.H"

FILE * trace;

VOID traceInst(VOID *ip) {
    fprintf(trace, "%p\n", ip);
}

VOID Instruction(INS ins, VOID *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)traceInst, IARG_INST_PTR, IARG_END);
}

int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
    return 0;
}
VOID Image(IMG img, VOID *v) {
    RTN mallocRtn = RTN_FindByName(img, "malloc");

    if (RTN_Valid(mallocRtn)) {
        RTN_Open(mallocRtn); // fetch insts in mallocRtn

        RTN_InsertCall(mallocRtn, IPOINT_BEFORE,
                        (AFUNPTR)Arg1Before,
                        IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);

        RTN_InsertCall(mallocRtn, IPOINT_AFTER,
                        (AFUNPTR)MallocAfter,
                        IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);

        RTN_Close(mallocRtn);
    }
}

Example: Malloc Trace
SimpleExamples/malloctrace.C

Example:

Example:

Malloc

Trace

VOID

RTN

mallocRtn

RTN_FindByName(img, "malloc")

if (RTN_Valid(mallocRtn))

RTN_Open(mallocRtn); // fetch insts in mallocRtn

RTN_InsertCall(mallocRtn, IPOINT_BEFORE,
                (AFUNPTR)Arg1Before,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);

RTN_InsertCall(mallocRtn, IPOINT_AFTER,
                (AFUNPTR)MallocAfter,
                IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);

RTN_Close(mallocRtn);

1st argument to malloc
returns wanted

1st return value (address
allocated)

before malloc’s entry

before malloc’s return

before malloc’s entry

before malloc’s return

before malloc’s entry
Instrumentation Philosophy

• Tools view instruction list of application instructions
• Users only insert function calls
• No general code modification ability for tools
• Try to close gap by inlining and optimization of instrumentation
• Still a gap in altering control flow
Pin tool inserts instrumentation when putting code in cache.
Just-in-time Instrumentation

Original code

1
2 3
4 5
6
7

Code cache

1’
2’
3’
5’
7’
6’

Compiler
JIT-based Instrumentation Features

- Handles mixed code and data, variable alignment, variable size instructions with 100% accuracy
- Maintains control at all times, change/observe anything
- Only instrument executed code
  - Database server code is 60Meg + shared libraries
- No special handling for shared libraries
- Handles dynamically generated code
- No dependence on compiler or binary format
  - Applies to instrumentation engine, but tools may need to access symbol information
- Trace based optimization of instrumentation
JIT Based Instrumentation

Drawbacks

• Time overhead 0% - 300%

• Hardware counters may not be give useful information
Probe-based instrumentation

- Overwrite original program with probe (branch) to reach dyninst-style trampolines
- Mostly execute original program, enter instrumentation via probes
- Subset of API
  - Started with very general capability
  - Replaced with straight jacket - wrap function calls to observe or alter behavior
- Near zero overhead and perturbation
- Relevant timing, hardware counter data
- Weak code and CFG discovery
  - Sufficient for tools that watch API usage
    - MPI trace analysis
    - Memory allocation errors
- Shares compiler & injector with JIT based instrumentation
Details

* IA32, Intel 64, IA64
* Linux, Windows, MacOs
* No charge
  - But not open source
* BSD-like license
  - no restrictions on use or redistribution
  - Instrumentation vm distributed as binary
  - Sample tools are open source
* Download it at [http://rogue.colorado.edu/Pin](http://rogue.colorado.edu/Pin)
  - 600 downloads/month
Pin Users

- Microprocessor development
  - Fast & easy to extend emulator
- Intel Software Quality and Performance Analysis Products
  - Emphasis on parallelism
- ISV
  - Software quality & performance tools
- University
  - research & education
Microprocessor Development

- Model performance of hardware that does not exist
- Instrumentation based tools are fast and easy to develop

- CMP$im – memory system performance modeling
- EMX – instruction emulation
- PinPoints, PinPlay – workload capture
CMP$im Features

- Use Pin to instrument all loads and store
- Fast Memory Characterization:
  - Single/multi-threaded workloads
  - 4-25 MIPS (100x-800x slow)
- Memory System Configurations:
  - Model private/shared caches
  - Model single/hyper-threaded cores
  - Model inclusive or non-inclusive caches
- Statistics:
  - Detailed instruction/cache statistics
  - View phase behavior of workloads

4-threaded, 4-core, CMP Multi-Level Cache Sharing:
- 4-threads per FLC
- 8-threads per MLC
- 16-threads per LLC
Sharing Phase Dependent & $f$ (cache size)

4 MB LLC  
16 MB LLC  
64 MB LLC

How Much Shared?

(a) SEMPHY

(b) SVM

4 Threaded Run:  
1 Thread   2 Thread   3 Thread   4 Thread
EMX - Instruction Emulation

- Architectural evaluation requires extensions to existing instruction sets
  - Debug compiler & libraries
  - Performance evaluation

- EMX pintool
  - Instrumentation replaces new instructions with emulation functions
  - Near native speed for everything else
  - Can use instrumentation to study programs that use extensions
Software Quality & Performance Analysis

Observe
  Pin instruments loads and stores, control flow

Analyze data

Present information
  Pin provides symbol/debug info

Many tools use both probe-based instrumentation for speed and JIT-based instrumentation for detailed analysis
Intel® VTune™ Performance Analyzer

- Call graph
- Other profilers
<table>
<thead>
<tr>
<th>Module (88)</th>
<th>Thread (88)</th>
<th>Function (88)</th>
<th>% in function</th>
<th>Calls (88)</th>
<th>Self Time (88)</th>
<th>Total Time (… Self Wait Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>clock</td>
<td>4.3%</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>exit</td>
<td>4.3%</td>
<td>1</td>
<td>2</td>
<td>46</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>free</td>
<td>4.3%</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>main</td>
<td>100.0%</td>
<td>1</td>
<td>10,763,282</td>
<td>10,763,578</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>mainCRTStartup</td>
<td>0.0%</td>
<td>1</td>
<td>7</td>
<td>10,763,861</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>malloc</td>
<td>2.0%</td>
<td>54</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>memmove</td>
<td>2.0%</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>memset</td>
<td>2.0%</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>printf</td>
<td>0.0%</td>
<td>3</td>
<td>0</td>
<td>196</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>strcpy</td>
<td>100.0%</td>
<td>49</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>prime_serial.exe</td>
<td>Thread_0(B20)</td>
<td>strlen</td>
<td>100.0%</td>
<td>105</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Vtune™ Call Graph Profile
Intel® Thread Checker

- Detects threading bugs
  - Data Races
  - Deadlocks
- Instruments loads, stores, threading API
Intel® Thread Checker

Memory read of number_of_primes at "2_openmp.cpp".14 conflicts with a prior memory write of number_of_primes at "2_openmp.cpp".14 (flow dependence)

```c
long factor = 3;
while ( number % factor ) factor += 2;
if ( factor == number )
    primes[ number_of_primes++ ] = number;

printf( "Found %d primes\n", number_of_primes );
```

```c
long factor = 3;
while ( number % factor ) factor += 2;
if ( factor == number )
    primes[ number_of_primes++ ] = number;

printf( "Found %d primes\n", number_of_primes );
```
How Does Thread Checker Work?

Thread 1

- Lock(L);
- n_of_p++
- Unlock(L);

Thread 2

- Lock(L);
- n_of_p++;
- Unlock(L);
How Does Thread Checker Work?

Use binary instrumentation

**Thread 1**

1. record lock(L)
2. Lock(L)
3. record read(n_of_p)
4. record write(n_of_p)
5. n_of_p++
6. record unlock(L)
7. Unlock(L)

**Thread 2**

1. record lock(L)
2. Lock(L)
3. record read(n_of_p)
4. record write(n_of_p)
5. n_of_p++
6. record unlock(L)
7. Unlock(L)
How Does Thread Checker Work?

Analysis reveals that segment 1 “happens before” segment 2. So, no data race problem.

Analysis based on [Lamport 1978]
How Does Thread Checker Work?

With no locks, neither segment “happens before” the other. So there is a data race!
• Find Contended Locks
  • Most Overhead
  • Largest Reduction in Parallelism

• Probe-based instrumentation of threading API’s
Using Thread Profiler

Profile View | Summary 1
Using Thread Profiler
Intel Trace Analyzer and Collector

- Collects MPI trace
- Correctness checking of usage
- Analysis for optimization
- Probe-based to instrument MPI calls
- JIT-based for precise call stacks
Intel Trace Analyzer and Collector
Cooperation

• Pin use of external components:
  – Cannot use GPL libraries, but LGPL OK
  – Symtab
  – Unwind
  – Code and CFG discovery (e.g. bloop)
  – Thread safe instrumentation tool runtime (libc)
    – No dependencies on system

• Intel contributions:
  – Pin is binary-only distribution
  – Difficult to provide binaries for non Intel ISA
  – XED IA32, Intel 64, AMD64 encoder/decoder/disassembler
  – Open source components? E.g. injector
XED encoder/decoder/disassembler

- Used in projects other than Pin
- Includes all public ISA extensions
- Correct
  - Only decode what really is an instruction
  - Get all the operands correct
  - read/write/conditional, size, register type, ...
  - Only encode well formed encode requests
- Fast, Small
- Thread safe
- Distributed as library in pin kit with manual
University Relations

• Close interaction with:
  – Harvard
  – MIT
  – U Colorado Boulder
  – U Virginia

• Internships
  – Self contained projects
    – Reliability, persistence
  – Adds support in pin for projects continued at university
    – Checkpoint/restart to support simulation
Summary

• Some instrumentation tools need full control and ability to instrument every instruction
  – JIT-based instrumentation

• Other tools need low overhead
  – Native execution with probes

• Both styles of instrumentation share functionality