cc by nc sa

Measurement: Instrumentation

Performance Evaluation of Computer Systems

Vojtěch Horký Peter Libič Petr Tůma

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University

2010 – 2021

Outline

1. Overview
2. Source Code Instrumentation
3. Bytecode Instrumentation
4. Machine Code Instrumentation
5. Instrumentation Overwriting Code
6. Instrumentation Translating Code

Instrumentation

Purpose

Instrumentation inserts measurement code (probes) into well defined program locations to facilitate data collection.

Examples of instrumentation:

- Static instrumentation in source files
- Dynamic instrumentation in object model
- Static or dynamic instrumentation in bytecode
- Static or dynamic instrumentation in machine code
- Static preparation in source files that allows dynamic instrumentation in machine code

Instrumentation

Collected Information

Information that is interesting but cannot be measured directly:

- Precise program traces
- Program state snapshots
- Function parameter values
- Anything else the program can observe ...

Applications

Useful for many dynamic analyses:

- Test coverage
- Race detection
- Taint tracking
- Origin tracking
- Memory usage checks

(cc) BY NC SA

Outline

1. Overview
2. Source Code Instrumentation
3. Bytecode Instrumentation
4. Machine Code Instrumentation
5. Instrumentation Overwriting Code
6. Instrumentation Translating Code

CC BY-NC-SA

Source Code Instrumentation

Benefits

- Relatively easy to insert
  - Source code is made to be modified
  - Automated tools sometimes struggle
- Program state naturally available
- Locations relevant to code structure

Challenges

- Instrumentation is subject to compiler optimization:
  - Optimized together with surrounding code
  - Can impact surrounding optimizations
  - Can be subject to code motion
- Conditional instrumentation possibly tricky
- Requires source code and compilation

Source Code Instrumentation

Manual

Programmer inserts measurement code into locations of interest.

- Fine grained placement control
- Tedious and possibly error prone

Assisted

Tool inserts measurement code into specified locations.

- Tool guarantees systematic coverage
- Specification of locations limited

For tool examples think about logging frameworks or macro processors.

Source Code Instrumentation Problems

Configurability

We want to have simple way to turn instrumentation on and off.

- Preprocessing
- Conditional calls

Think what a modern runtime will do with the following:

```
log.debug ("App␣results␣are␣'" + results.toString () + "'.");
log.debug ("App␣results␣are␣'{}'.", results);
if (log.isDebugEnabled ()) {
  log.debug ("App␣results␣are␣'" + results.toString () + "'.");
}
```

Source Code Instrumentation Problems

Reliability

We need reliable association between instrumentation and application code.

Think what a modern compiler will do with the following:

```
int sqrt_counter = 0;
inline double counted_sqrt (double x) {
  sqrt_counter ++;
  return (sqrt (x));
}
```

This is the square root function disassembly:

```
sqrt:   pxor    xmm1,xmm1       // Set xmm1 to zero
        ucomisd xmm1,xmm0       // Compare argument to zero
        ja      fail            // Negative argument check
        sqrtsd  xmm0,xmm0       // Compute square root
        ret                     // Register passing
```

Outline

1 Overview

2 Source Code Instrumentation

3 Bytecode Instrumentation

4 Machine Code Instrumentation

5 Instrumentation Overwriting Code

6 Instrumentation Translating Code

cc by sa

Bytecode Instrumentation

Benefits

- Still relatively easy to insert
  - Bytecode contains much metadata
  - Compilation produces predictable bytecode
- Program state still naturally available
- Can be done at runtime without sources

Challenges

- Requires tools
- Not all languages and environments have bytecode
- Locations in bytecode possibly different from code structure

For tool examples consider ASM or DiSL.
Also aspect oriented frameworks such as AspectJ.

CC BY-NC-SA

ASM

Library for bytecode manipulation.
Main features of ASM are:

- Core API based on visitor design pattern
  - `ClassReader` to generate events from class file
  - `ClassWriter` to generate class file from events
  - Transformations implemented as event pipes
  - Adapters for predefined transformations
- Tree API for in memory class file representation
  - Can build representation from Core API events
  - Can generate Core API events from representation

`http://asm.ow2.io`

cc by sa

Aspect Oriented Programming

Idea

What if we could express independent concerns
by separate code fragments ?

- Logging
- Transactions
- Authorization

concern	Program feature that stands apart from other features.
join point	Program location where concern code resides.
pointcut	Specification of a set of join points.
advice	Code inserted at pointcut.
weave	Insert advice.

Obviously AOP can be used to insert measurement instrumentation.
Some pointcut specifications can introduce significant perturbation.

AspectJ

Aspect oriented programming framework for Java.

Main features of AspectJ are:

- Byte code instrumentation at compile time and load time
- Declarative language for defining instrumentation points
- Instrumenting code written in Java

`http://www.eclipse.org/aspectj`

Instrumentation with AspectJ

```
aspect Measurement {

    // Select all executions of methods of class Main.
    pointcut allMainMethods (): execute (* Main.* (..));

    // Attach an around advice that measures time.
    Object around (): allMainMethods () {
        long timeBefore = System.nanoTime ();
        Object result = proceed ();
        long timeAfter = System.nanoTime ();
        System.out.println (timeAfter - timeBefore);
        return (result);
    }
}
```

AspectJ Join Points I

Join points are:

- call and execution of a method or a constructor,
- execution of an exception handler,
- execution of a static initializer,
- read or write access to a field,
- execution of an advice.

```
execution (int SomeClass.someMethod (int))
execution (String *.get*Name (..)
call (* AnotherClass.* (String))
call (*.new (long))
handler (RemoteException+)
get (int *.counter)
```

AspectJ Join Points II

Join points can be further constrained by:

- presence of an annotation,
- location within a class or a method,
- actual type of the current object, called object, arguments,
- control flow selected by particular pointcut,
- boolean expression.

```
this (SomeClass)
target (AnotherClass)
args (int, int, int, int)
cflow (SomePointcut)
within (SomeClass)
```

cc by sa

AspectJ Pointcuts I

Pointcuts combine specifications of join points
using standard operators &&, ||, !.

```
pointcut callToSomeClassFromMain ():
    within (Main) && target (SomeClass+);

pointcut nonRecursiveCallToSomeClass ():
    call (* SomeClass.* (..)) && !within (SomeClass)
```

CC BY-SA

AspectJ Pointcuts II

Pointcuts can make accessible variables in their context:

- current object,
- target object,
- arguments.

```
pointcut callToSomeClass (SomeClass o):
    call (* SomeClass.* (..)) && target (o);

pointcut namingSomething (String name):
    call (void *.set*Name (String)) && args (name);
```

AspectJ Advice

Advice can be associated with join point:

- before the join point,
- after the join point
  - when it returns normally,
  - when it throws an exception,
- around the join point.

```
before SomePointcut ():
    Object [] arguments = thisJoinPoint.getArgs ();
    for (Object argument : arguments) {
      System.out.println (argument);
    }

Object around AnotherPointcut ():
    return (proceed ());
```

More AspectJ Features

Aspects can include declarations across types:

- declare new fields,
- declare new methods,
- declare new constructors,
- introduce new parents,
- introduce new interfaces.

```
private interface HasCounter {}
declare parents:
    (SomeClass || AnotherClass) implements HasCounter;
private long HasCounter.executionCount = 0;
before (HasCounter o):
    execution (* *.* (..)) && this (o) {
    o.executionCount ++;
}
```

CC BY-SA

Bytecode Instrumentation Problems

Consider

Inserted instrumentation executes in the same virtual machine as program.

Can this cause any problems ?

- Deadlock between analysis and application
- State corruption inside application code
- Arbitrary assumptions in virtual machine
- Bytecode verification failures
- Shared reference handlers
- Coverage approximation
- ...

Based on Kell et al.: The JVM is Not Observable ... doi:10.1145/2414740.2414747

CC BY NC SA

Program State Corruption

Coverage

Analyses may require total code coverage:

- Memory allocation tracking (leaves important)
- Taint tracking (and any other data flow)
- Reliable race detection
- ...

What happens if we try to instrument every class of the application ?

(CC BY SA icons)

Program State Corruption

Coverage

Analyses may require total code coverage:

- Memory allocation tracking (leaves important)
- Taint tracking (and any other data flow)
- Reliable race detection
- ...

What happens if we try to instrument every class of the application ?

This includes `Object`, `String`, `System` classes.
These will likely be used by instrumentation and analysis code too.
It is (too) easy to run into infinite recursion or state corruption problems.

Dynamic Bypass Pseudocode

```
static boolean instrumentationOnline = false;

// Instrumentation snippet
if (!instrumentationOnline) {
  instrumentationOnline = true;

  // Actual instrumentation here

  instrumentationOnline = false;
}
```

Simple with single-threaded programs,
can be tricky with multiple threads.

(cc) BY NC SA

DiSL

A bytecode instrumentation framework with emphasis on coverage.

- Language and framework for Java bytecode instrumentation
- Similar to aspects, but read only and with more control
- Allows to write instrumentation snippets directly in Java
- Instrumentation points are selected using annotations
- Dynamic bypass is provided automatically

`http://disl.ow2.org`

Example Instrumentation Snippet

```
@Before (
    marker = BodyMarker.class,
    scope = "TargetClass.print␣(boolean)",
    order = 8)
public static void precondition () {
    System.out.println ("Precondition␣!");
}
```

snippet	Code inserted as instrumentation.
marker	Specification of location to instrument.
scope	Specification of classes to instrument.
guard	Instrumentation condition.

CC BY-NC-SA

DiSL Architecture

Instrumentation Server

Standalone server responsible for creating instrumented classes.

- Standalone to minimize perturbation
- Instrumentation using ASM
- Optimizations

Application Client

Java virtual machine executing the instrumented application.

- JVMTI agent to intercept class loading process
- Remote communication with instrumentation server
- Also executes whatever instrumentation code is inserted

Outline

1. Overview
2. Source Code Instrumentation
3. Bytecode Instrumentation
4. Machine Code Instrumentation
5. Instrumentation Overwriting Code
6. Instrumentation Translating Code

(cc) BY NC SA

Machine Code Instrumentation

Benefits

- Machine code (almost) always available
- Looking at code in very fine resolution

Challenges

- Machine code difficult to analyze
  - Mixing code and data
  - Variable length instructions
  - Very far from source code structure
- Inserting extra code difficult
  - No space for extra instructions
  - Register state must be preserved
- Some patterns complicate things
  - Auto generated or self modifying code
  - Computed branch targets

Recognizing Machine Code

DUMP

```
.. 04 0A 11 1F 11 11 ..
```

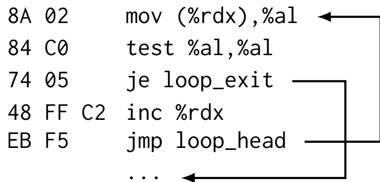
AS CODE

```
...
04 0A    add $0xa,%al
11 1F    adc %ebx,(%rdi)
11 11    adc %edx,(%rcx)
...
```

AS BITS

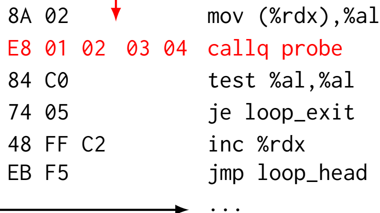
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	1	1
1	0	0	0	1
1	0	0	0	1

Shifting Machine Code



BEFORE PATCH

AFTER PATCH



Machine Code Instrumentation

Available tools use combinations of many techniques:

- Overwrite instrumentation locations
- Instrument only prepared locations
- Use dynamic translation

Example tools: DTrace, KProbes, PIN, Valgrind, DynamoRIO.

Outline

1. Overview
2. Source Code Instrumentation
3. Bytecode Instrumentation
4. Machine Code Instrumentation
5. Instrumentation Overwriting Code
6. Instrumentation Translating Code

CC BY-NC-SA

Overwriting Instructions

Instrumentation can be inserted by overwriting the instruction(s) at the target location.

Challenges

- Must overwrite only single location
  - As little as single byte on variable opcode length architectures
  - Single instruction on constant opcode length architectures
- Overwrite must be an atomic operation
- Original instruction must be replayed

- Specialized instructions
  - Intel `INT 3` opcode is single byte `0xCC`
  - Translates into `SIGTRAP` on Linux
  - But also writes on stack
- Hardware support
  - Explicit control transfers or barriers sometimes needed
- Trampolines

See Chamith et al.: Living On The Edge ... doi:10.1145/2908080.2908084

CC BY SA

Instrument Prepared Locations

Instrumentation can be inserted at locations
that were previously prepared for such use.

Challenges

- Identifying suitable locations
- Low overhead when not instrumented

- Prologues
  - Compilers can generate suitable function prologue
- Exported symbols
  - Reasonable location to expect instrumentation at
  - Often called through relatively standard code (PLT)
- Specialized instructions
  - Intel `NOP` has opcode variants for up to nine bytes
  - Atomic updates for long variants require some care

cc by sa

Linux Kernel Function Tracer

System for tracing calls to kernel functions:

- Instrumentation locations prepared by compiler
  `/sys/kernel/debug/tracing/available_filter_functions`
- Probes dynamically disabled and enabled
- Accessed through debug file system
  `/sys/kernel/debug/tracing/available_tracers`
  `/sys/kernel/debug/tracing/current_tracer`
  `/sys/kernel/debug/tracing/trace`

See Linux kernel `Documentation/trace/ftrace.rst`

Linux Static Kernel Trace Points

System for tracing predefined kernel events of interest:

- Inserted by kernel developers into appropriate locations
  `/sys/kernel/debug/tracing/available_events`
  `/sys/kernel/debug/tracing/events` tree
- Probes dynamically disabled and enabled
- Accessed through debug file system
  `/sys/kernel/debug/tracing/set_event`
  `/sys/kernel/debug/tracing/trace`

See Linux kernel `Documentation/trace/tracepoints.rst`

Linux Kernel Probes

Interfaces that allow to instrument:

- Any single instruction in kernel (Kprobes)
- Any function entry and exit in kernel (Return Probes)

See Linux kernel `Documentation/trace/kprobes.rst`

(cc) (by) (nc) (sa)

Kprobes

Replace target instruction with breakpoint instruction

- Breakpoint instruction is single byte
- Control transfer similar to interrupt
- Jumps can also be used (faster)

Execute probe code

- Registers saved as in other interrupts
- Handler for both pre code and post code

Single step the replaced instruction

- Must be done inside interrupt handler
- Requires understanding all instructions

Return Probes

Replace function entry with Kprobe

- Kprobe saves original return address
  - Limit on concurrently executing functions with return probes
  - Invocations exceeding limit are counted but not intercepted
- Kprobe modifies function return address to point to trampoline
- Trampoline is instrumented with another Kprobe at system boot time

Execute function code

- Function executes normally
- On return control is passed to trampoline Kprobe

Linux Static User Mode Probe Points

System for marking predefined user mode locations of interest:

- Inserted by application developers into appropriate locations

```
#include <sys/sdt.h>
STAP_PROBE (app, location)
STAP_PROBE1 (app, location, arg1)
STAP_PROBE2 (app, location, arg1, arg2)
...
```

- Probes compile into
  - A NOP instruction in the probe location
  - A probe record in the ELF `stapsdt` note section

The `perf` tool recognizes user mode probe points after scanning the binary with `perf buildid-cache`.

See `man stapprobes`

CC BY-SA

Linux User Mode Probes

System for tracing dynamically instrumented user mode code locations:

- Instrumentation code inserted by kernel on code load
- Controlled through debug file system
  `/sys/kernel/debug/tracing/uprobe_events`
  `echo "<type> <file>:<offset> [arglist]"`

Support included in the perf tool:

```
> perf probe -x <file> --funcs
> perf probe -x <file> --line <func>
> perf probe -x <file> --vars <func>:<line>
> perf probe -x <file> <func>:<line> [<var> ...]
```

See Linux kernel `Documentation/trace/uprobetracer.rst`

CC BY SA

Linux Extended Berkeley Packet Filters

Framework for securely injecting code into kernel.

Code

Injected code written in limited bytecode:

- RISC style instructions
- Limit on instruction count (1M)
- Limit on branching (no loops)
- Static memory access checks

Bytecode supported by LLVM backend.

Maps

Data export through maps:

- In kernel key value stores
- Both global and per processor

See `man bpf`

cc by sa

Outline

1. Overview
2. Source Code Instrumentation
3. Bytecode Instrumentation
4. Machine Code Instrumentation
5. Instrumentation Overwriting Code
6. Instrumentation Translating Code

Dynamic Instrumentation

Instrumentation can be inserted by translating code during execution.

Challenges

- Identifying code to translate
- Keeping execution overhead reasonably low
- Making translation invisible to application

- Code recognized during execution
  - Anything executed must be code
  - Translate in units of basic blocks
  - Chain basic blocks from hot paths into traces
  - Cache and reuse translations during execution
- Instrument inside basic block discovery notification
- Interface to internal code representation
  - Close to binary form in “Copy and Annotate” (PIN, DynamoRIO)
  - Close to compiler IR in “Disassemble and Resynthesize” (Valgrind)

PIN

Intel dynamic binary instrumentation tool.
`http://software.intel.com/en-us/articles/pintool`

- C and C++ API
- Provides multiple instrumentation points such as Routine (RTN), Image (IMG), Instruction (INS)
- Instrumentation snippet is normal C/C++ code
- Operates in JIT mode or probe mode
- Supports x86

PIN Tool Example

```
int main (int argc, char * argv []) {
    if (PIN_Init (argc, argv)) exit (1);
    INS_AddInstrumentFunction (CountInstruction, 0);
    PIN_StartProgram ();
}

VOID CountInstruction (INS ins, VOID * v) {
    INS_InsertCall (ins, IPOINT_BEFORE, (AFUNPTR) DoCount, IARG_END);
}

VOID DoCount () { ... }
```

... run with `pin -t pintool.so -- command`

Based on code from Intel PIN examples

cc by nc sa

Valgrind

Open source dynamic binary instrumentation tool.
`http://valgrind.org`

- C API
- Compiler style intermediate representation (VEX)
- Instrumentation implemented as VEX manipulation
- Targets heavyweight instrumentation
- Supports x86, ARM, PPC, MIPS ...

Valgrind VEX Example

Original Instruction

```
add eax,ebx
```

```
------ IMark(0x123456, 2, 0) ------
# Connects VEX to original code address and length

t3 = GET:I32(8)   # Guest state offset 8 is EAX
t2 = GET:I32(20)  # Guest state offset 20 is EBX
t1 = Add32(t3,t2)
PUT(8) = t1

# Does not show flags and program counter updates
```

Based on code from Valgrind headers

Valgrind VEX Example

Original Instruction

```
add [eax+4],edx
```

```
------ IMark(0x123456, 4, 0) ------
# Connects VEX to original code address and length

t3 = Add32(GET:I32(8),0x4:I32)  # Non flattened
t2 = LDle:I32(t3)               # Little endian load
t1 = GET:I32(16)                # Guest state offset 16 is EDX
t0 = Add32(t2,t1)
STle(t3) = t0                   # Little endian store

# Does not show flags and program counter updates
```

Based on code from Valgrind headers

(CC BY-NC-SA)