Lund University

EDAP15: Program Analysis

Dynamic Program Analysis 2

Christoph Reichenbach

Welcome back!

- No quiz 12 or 14
- Office hours tomorrow from 15:30–16:30
- Oral exams 17th and 18th of March, registration opens on Friday (Moodle)
- **lab 4** on Thursday
  - Short lab
  - e-mail results to Christoph
  - Make sure you have `podman` or `docker` installed
- Catch-up lab next **Tuesday** in E:1407 from 13:15–15:00
  - **Final chance to present labs 2 and 3**

Questions?

Execution Traces

Source program

```
(0)fun f(arg : int) = {
(1)  return arg + 1;
(2)}
(3)fun g(x : int) = {
(4)  return x - 1;
(5)}

(6) fun main() = {
(7)   var x := f(1);
(8)   var y := f(100);
(9)   while x != y {
(10)    x := x + 1;
(11)    y -= x;
(12)  }
(13)}
```

Trace

6
7
0
1
8
0
1
9
10
11
9
10
11
9
10
11
⋮

Probing

```
fun f(arg :  int) = {
  return arg + 1;
}

fun main() = {
   var x := f(1);
   var y := f(100);
   while x != y {
     x := x + 1;
     y -= x;
   }
}
```

log(arg)

log(y)

Measurements

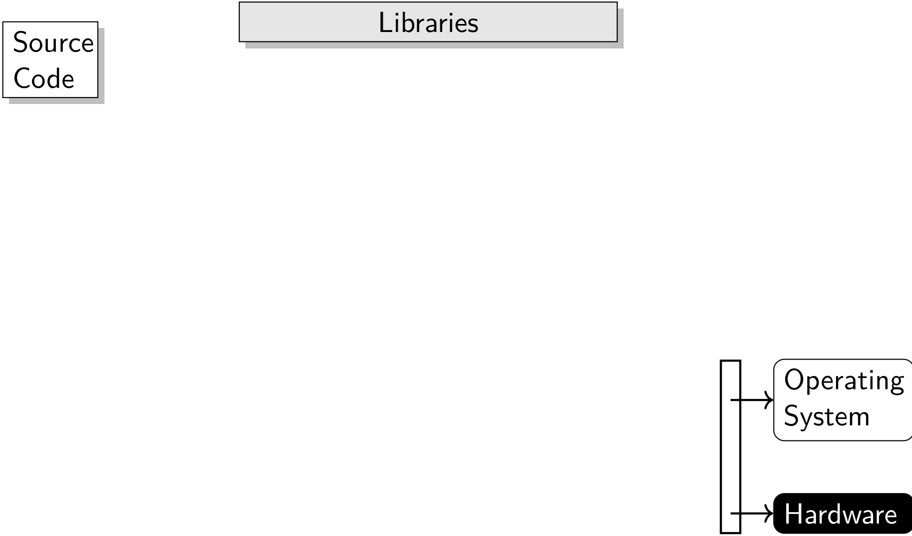
arg	1	Sample(arg)
arg	100	
y	98	Sample(y)
y	94	
y	89	
y	83	
y	76	
y	68	
y	59	
y	49	
y	38	
y	26	
y	13	

- Measure behavior with **Probes**
  - May need *Instrumentation* in program / runtime / hardware
- Probes triggered at certain **Events**
- Measure specific **Characteristics**
- Collect **Samples** of individual **Measurements**

General Data Collection

- *Probes*: How we measure
- *Events*: When we measure
- *Characteristics*: What we measure
- *Measurements*: Individual observations
- *Samples*: Collections of measurements

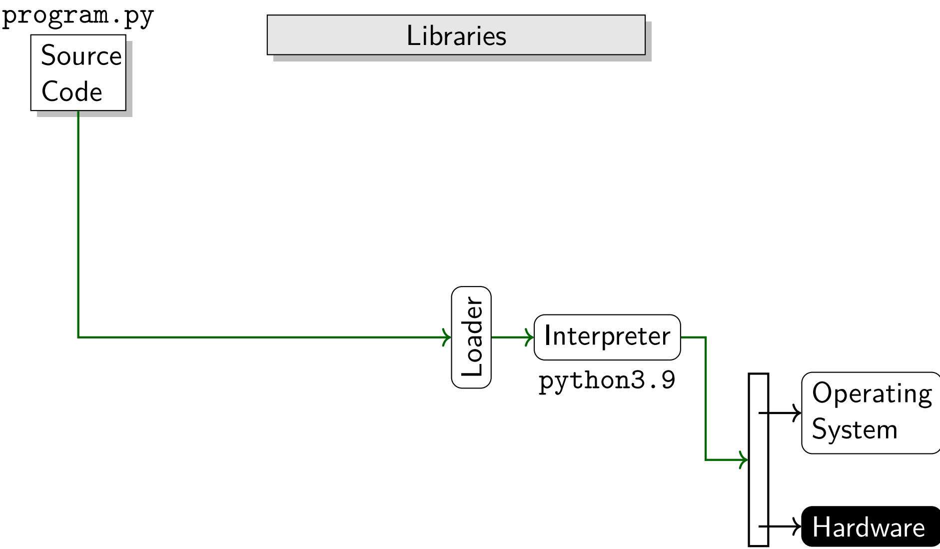
Probes



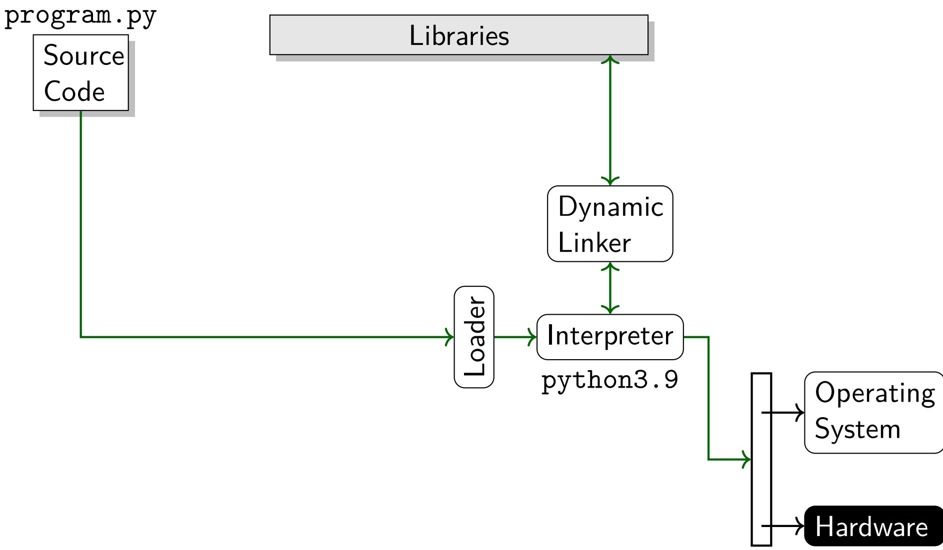
Probes


program.py
Source Code
Libraries
Operating System
Hardware

Probes


program.py
Source Code
Libraries
Loader
Interpreter
python3.9
Operating System
Hardware

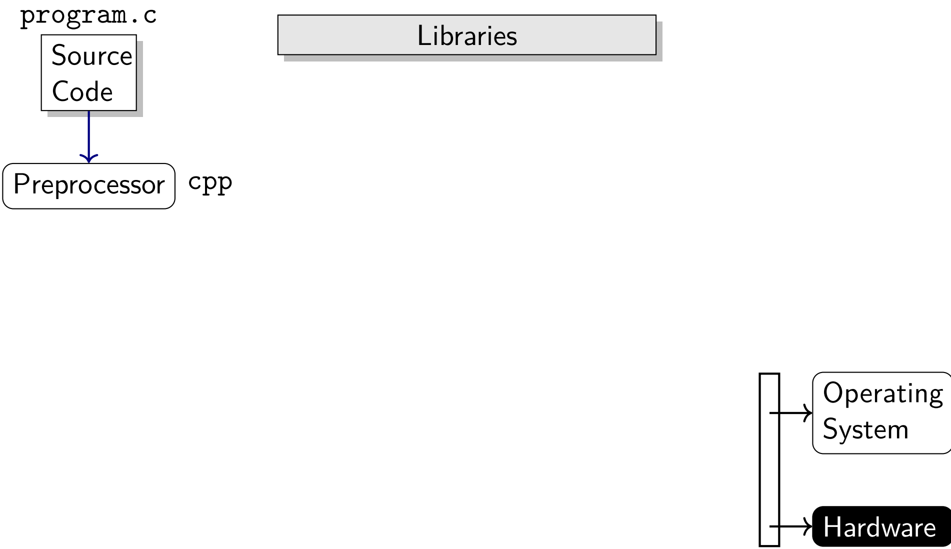
Probes


program.py
Source Code
Libraries
Dynamic Linker
Loader
Interpreter
python3.9
Operating System
Hardware

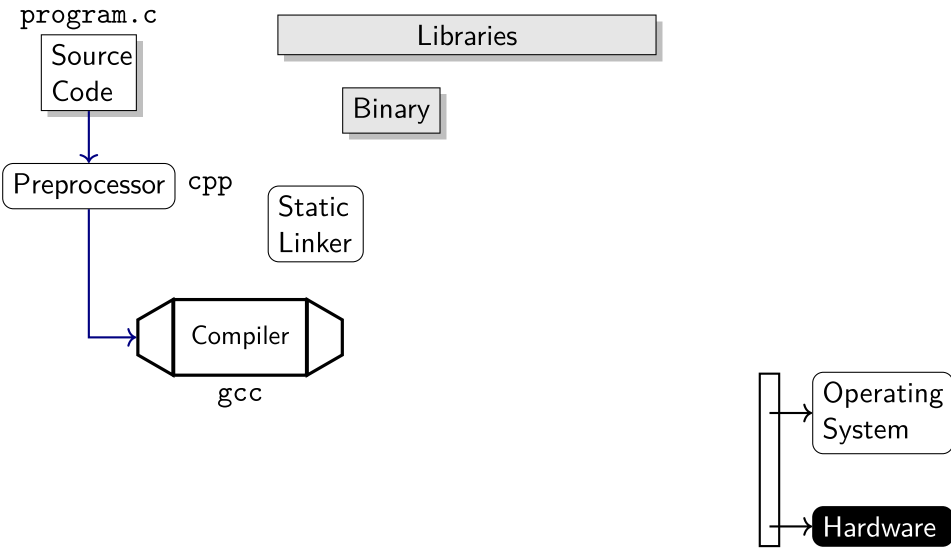
Probes


program.c
Source
Code
Libraries
Operating
System
Hardware

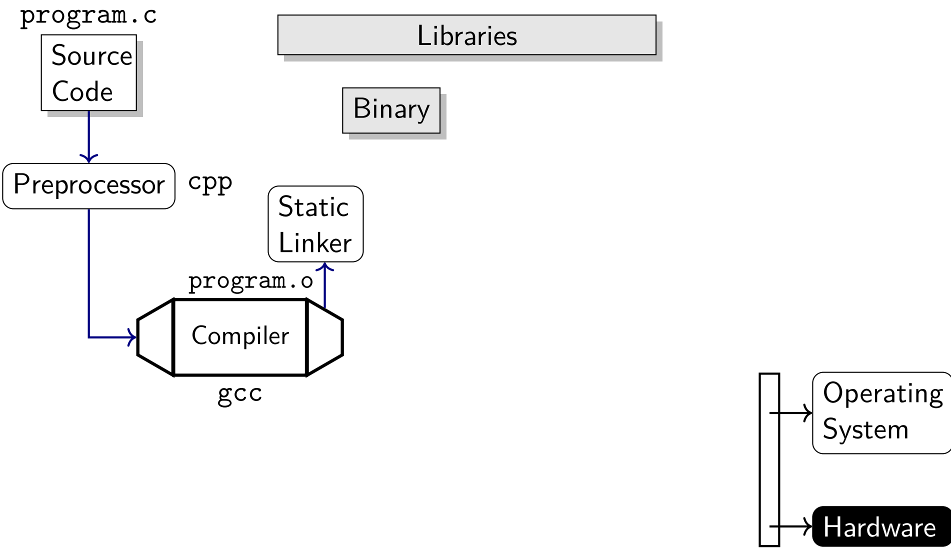
Probes


program.c
Source Code
Libraries
Preprocessor
cpp
Operating System
Hardware

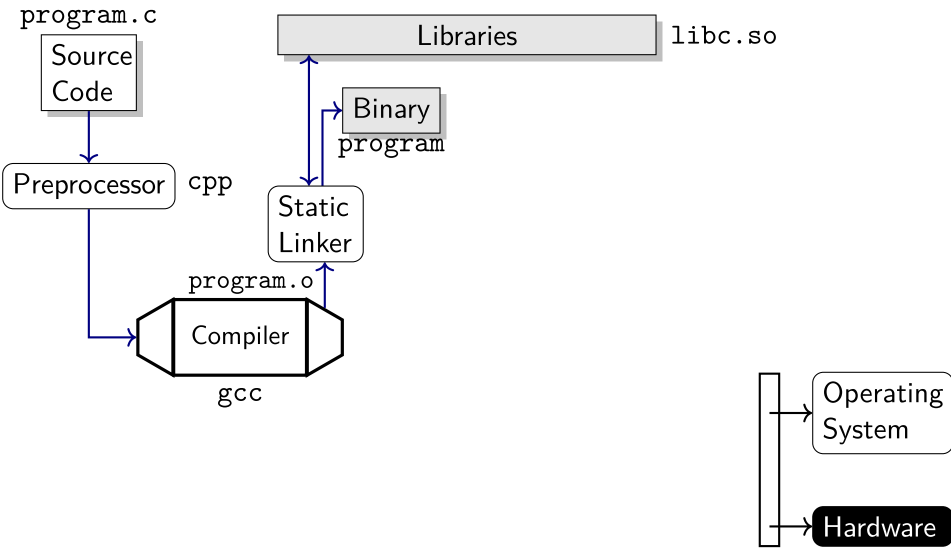
Probes


program.c
Source Code
Libraries
Binary
Preprocessor
cpp
Static Linker
Compiler
gcc
Operating System
Hardware

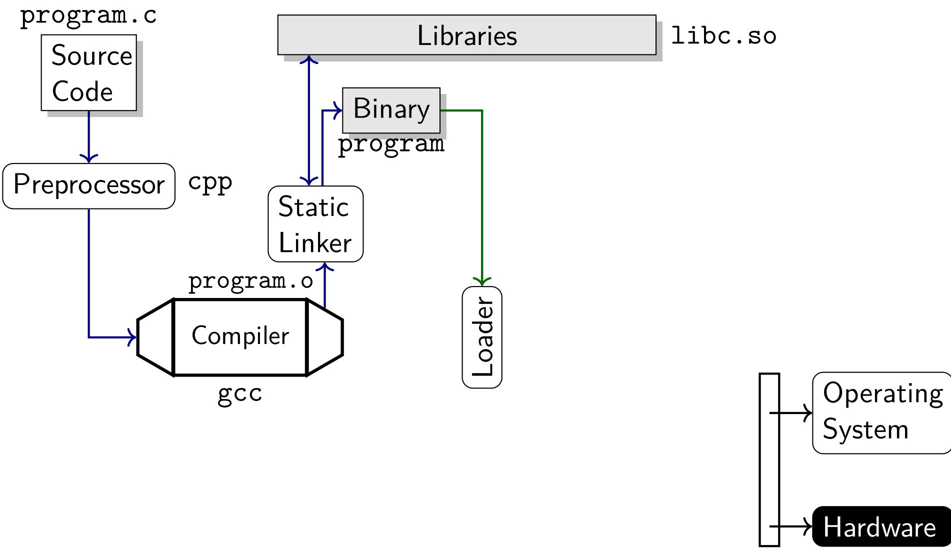
Probes


program.c
Source Code
Libraries
Binary
Preprocessor
cpp
Static Linker
program.o
Compiler
gcc
Operating System
Hardware

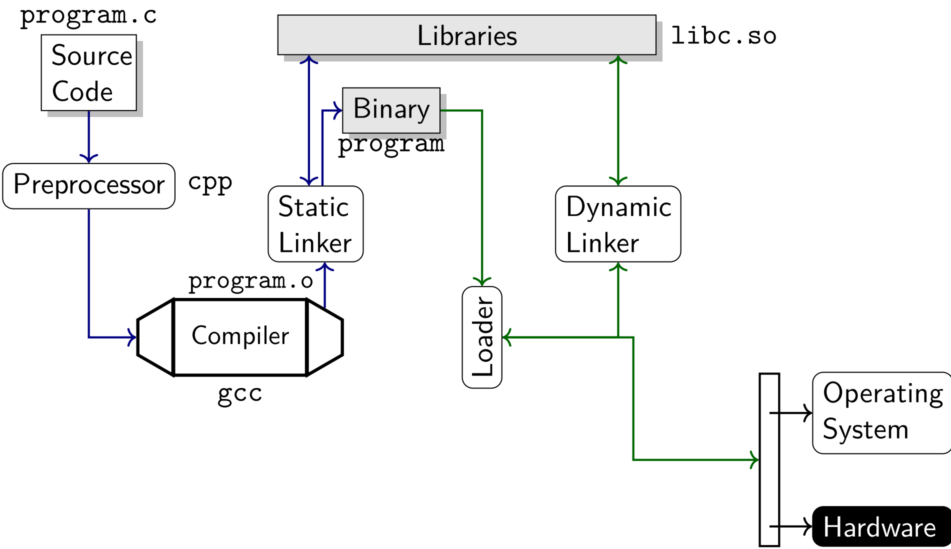
Probes


program.c
Source Code
Libraries
libc.so
Binary
program
Preprocessor
cpp
Static Linker
program.o
Compiler
gcc
Operating System
Hardware

Probes

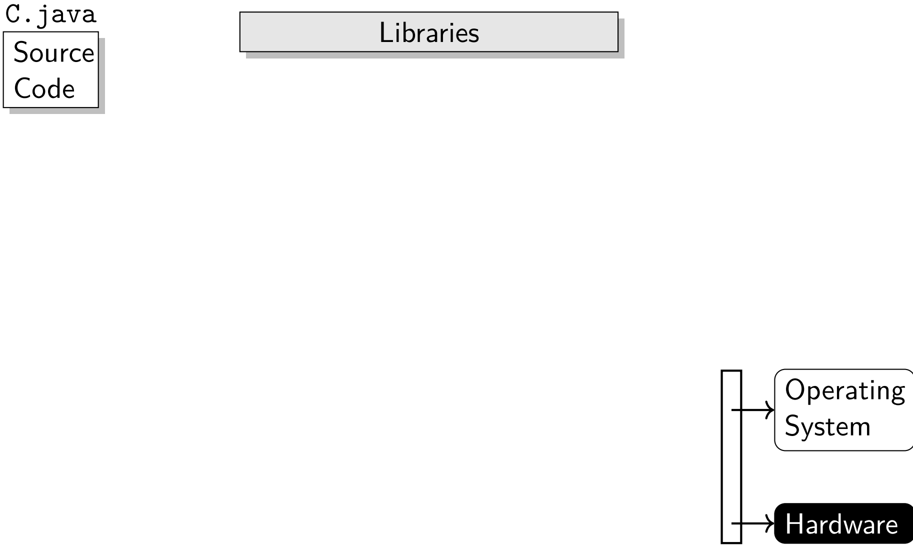

program.c
Source Code
Libraries
libc.so
Binary
program
Preprocessor
cpp
Static Linker
program.o
Compiler
gcc
Loader
Operating System
Hardware

Probes

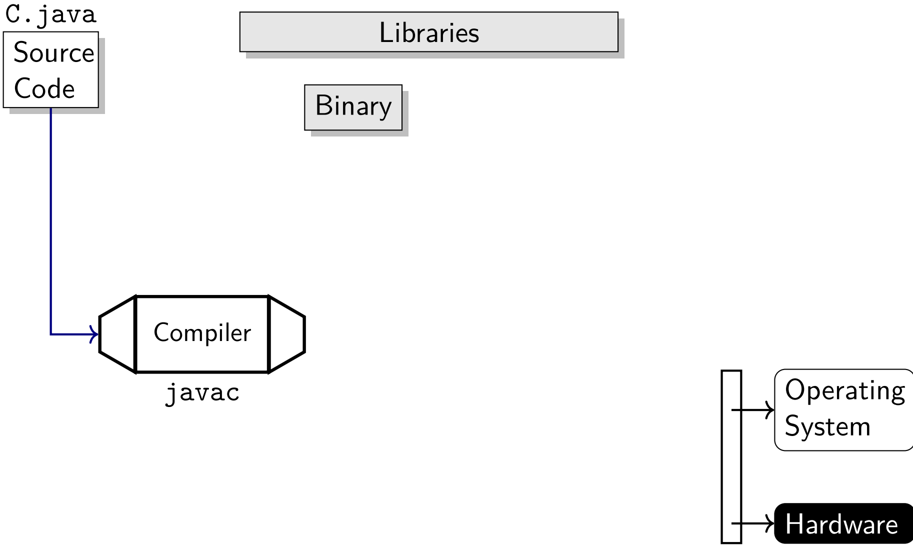


program.c
Source Code
Preprocessor
cpp
Compiler
gcc
program.o
Static Linker
Libraries
libc.so
Binary
program
Loader
Dynamic Linker
Operating System
Hardware

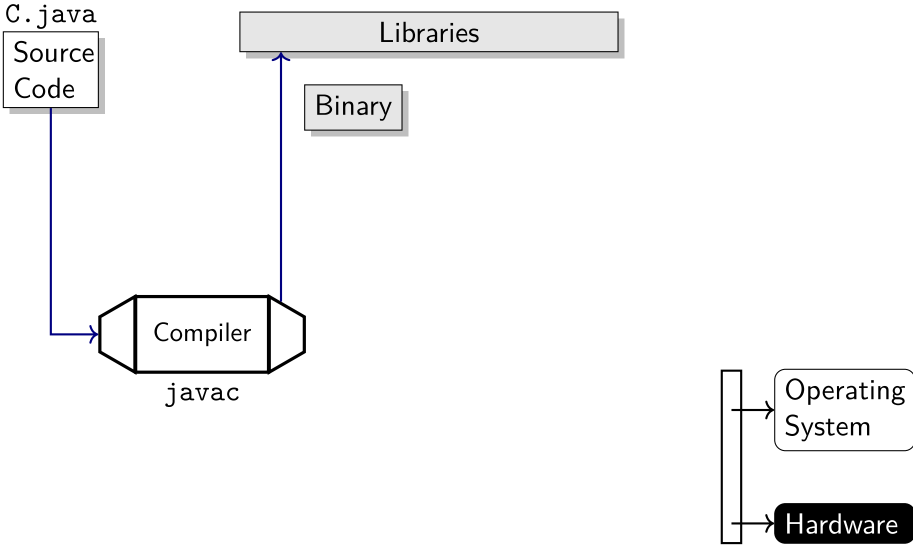
Probes



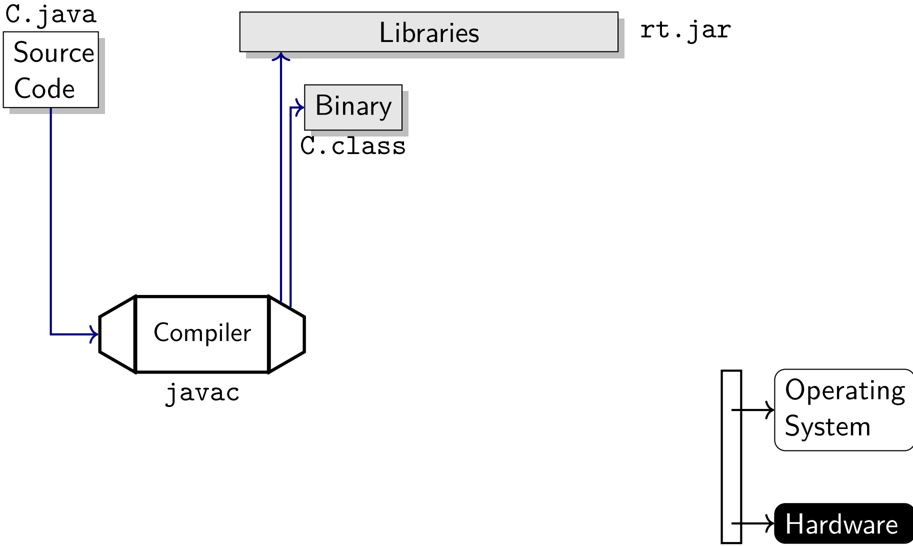
Probes


C.java
Source Code
Libraries
Binary
Compiler
javac
Operating System
Hardware

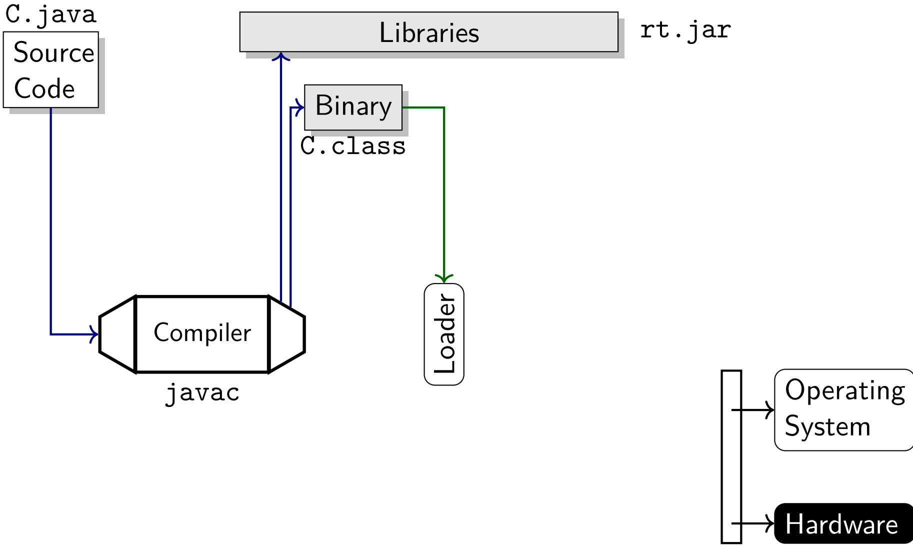
Probes


C.java
Source Code
Libraries
Binary
Compiler
javac
Operating System
Hardware

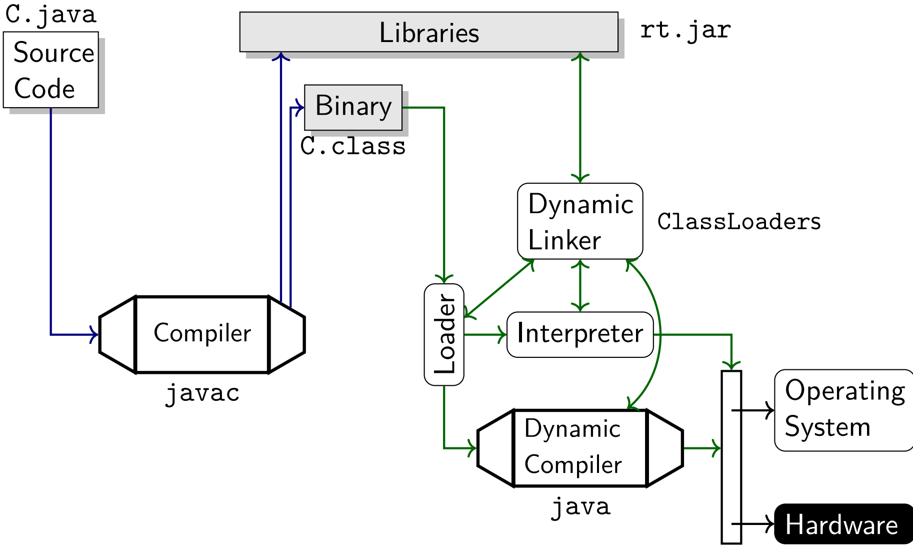
Probes


C.java
Source Code
Libraries
rt.jar
Binary
C.class
Compiler
javac
Operating System
Hardware

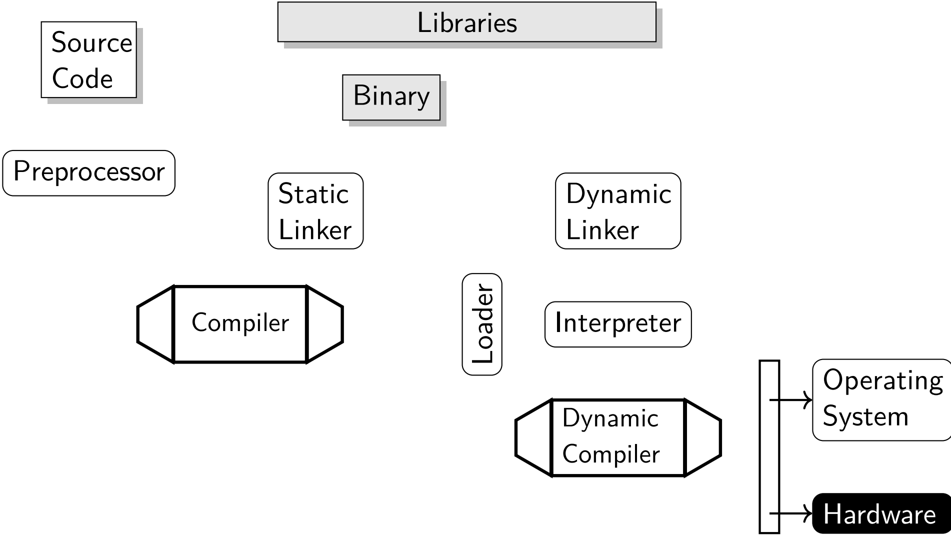
Probes


C.java
Source Code
Libraries
rt.jar
Binary
C.class
Compiler
javac
Loader
Operating System
Hardware

Probes

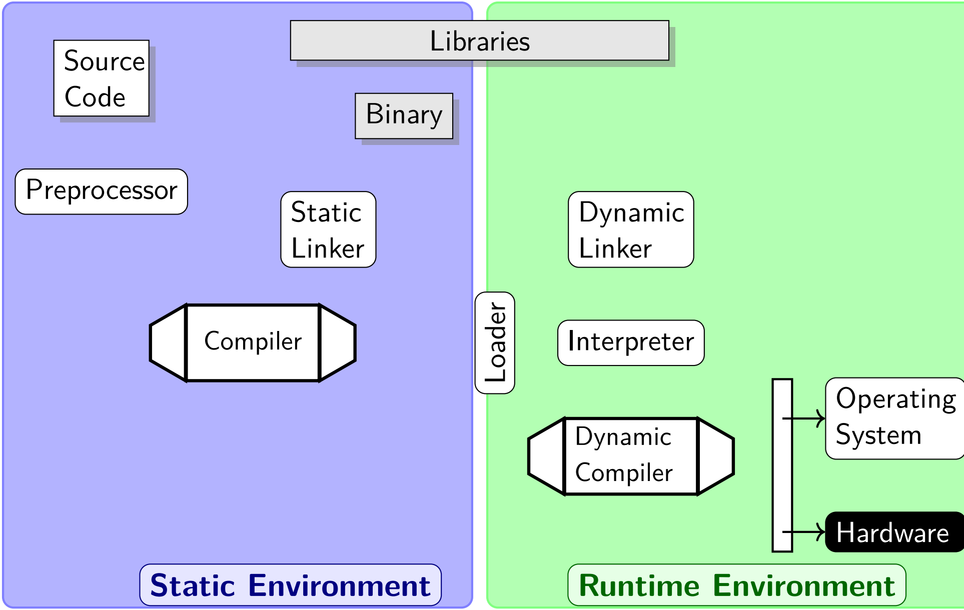

C.java
Source Code
Libraries
rt.jar
Binary
C.class
Dynamic Linker
ClassLoaders
Compiler
javac
Loader
Interpreter
Dynamic Compiler
java
Operating System
Hardware

Probes

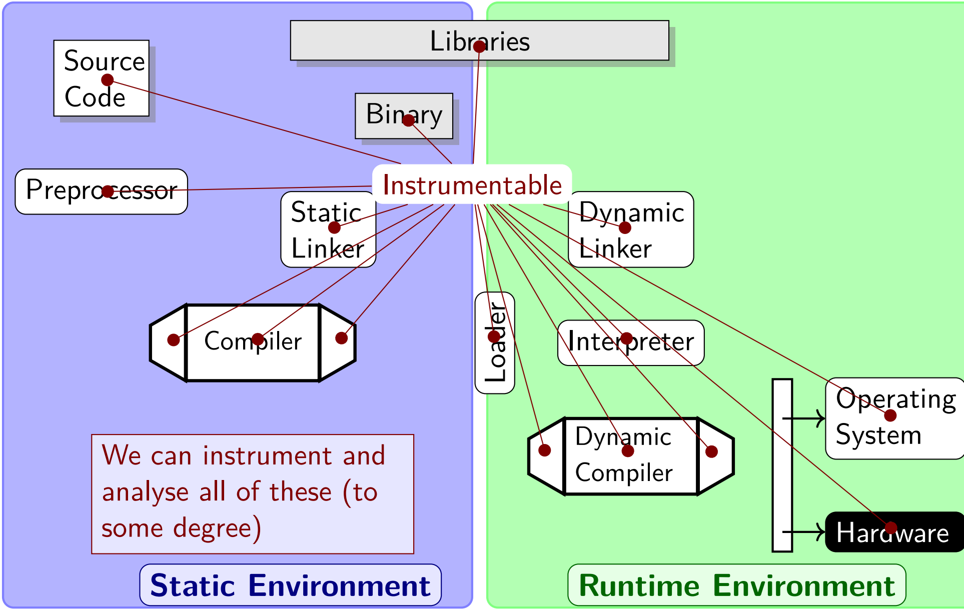


Libraries
Source Code
Binary
Preprocessor
Static Linker
Dynamic Linker
Compiler
Loader
Interpreter
Dynamic Compiler
Operating System
Hardware

Probes


Libraries
Source Code
Binary
Preprocessor
Static Linker
Dynamic Linker
Compiler
Loader
Interpreter
Operating System
Dynamic Compiler
Hardware
Static Environment
Runtime Environment

Probes



Libraries
Source Code
Binary
Preprocessor
Instrumentable
Static Linker
Dynamic Linker
Compiler
Loader
Interpreter
Operating System
Dynamic Compiler
Hardware
We can instrument and analyse all of these (to some degree)
Static Environment
Runtime Environment

Events

- Subroutine call
- Subroutine return
- Memory access (read or write or either)
- System call
- Page fault

...

Characteristics

- *Value*: What is the type / numeric value / ...?
- *Counts*: How often does this event happen?
- *Wallclock times*: How long does one event take to finish, end-to-end?

Derived properties:

- *Frequencies*: How often does this happen
  - Per run
  - Per time interval
  - Per occurrence of another event
- *Relative execution times*: How long does this take
  - As fraction of the total run-time
  - As fraction of some surrounding event

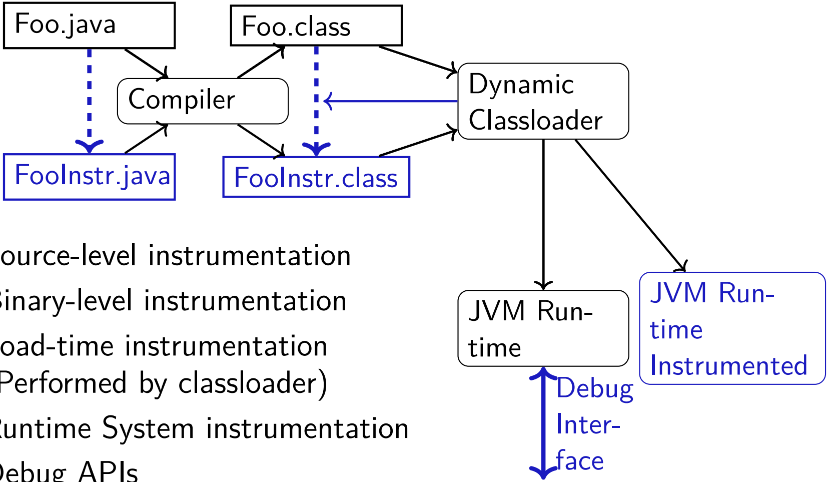
Probes

- **Probes**: devices for measuring property of interest
  - Software probe: code artefact
  - Hardware probe: physical device
- CPU, OS kernel etc. come with probes preinstalled
  - Generally need to be flipped on
- Want to probe custom location / property:
  - **Instrumentation**: insert new probes

Gathering Dynamic Data

- **Instrumentation and Software Probes**
- Simulation
- Hardware Probes

Gathering Dynamic Data: Java



- Source-level instrumentation
- Binary-level instrumentation
- Load-time instrumentation (Performed by classloader)
- Runtime System instrumentation
- Debug APIs

Comparison of Approaches

- **Source-level instrumentation**:
  - \+ Flexible
  - − Will miss events for which we lack source code
  - − Watch out for: control flow, name capture
- **Binary-level instrumentation**:
  - \+ Flexible
  - − Must handle machine code encoding/decoding, hardware-dependent
  - − Watch out for: run-time code generation (JITs etc.), dynamic loading
- **Load-time instrumentation**:
  - "Add-on" for source-level / binary-level instrumentation
  - \+ Can handle even unknown code
  - − Requires run-time support, may clash with custom loaders
- **Runtime system instrumentation**:
  - Especially for interpreters, JIT-compiled languages
  - \+ Can see "hidden" events (gc, JIT, . . .)
  - − Labour-intensive and error-prone
  - − Becomes obsolete quickly as runtime evolves
- **Debug APIs**:
  - \+ Typically easy to use, relatively well-documented
  - − Limited capabilities

Instrumentation Tools

	C/C++ (Linux)	**Java**
Source-Level	C preprocessor, DMCE	ExtendJ
Binary Level	`pin`, `llvm`	`soot`, `asm`, `bcel`, AspectJ, ExtendJ
Load-time	?	Classloader, AspectJ
Debug APIs	`strace`	JVMTI

- Low-level data gathering:
  - Command line: `perf`
  - Time: `clock_gettime()` / `System.nanoTime()`
  - Process statistics: `getrusage()`
  - Hardware performance counters: PAPI

Practical Challenges in Instrumentation

- *Measuring*:
  - Need access to relevant data
    (e.g., Java: source code can't access JIT internal)
  - May need to insert **software probes** (measuring device)
- *Representing (optional)*:
  - Store data in memory until it can be emitted (optional)
  - May use memory, execution time, *perturb measurements*
- *Emitting*:
  - Write measurements out for further processing
  - May use memory, execution time, *perturb measurements*

Summary

- Different **instrumentation strategies**:
  - Instrument **source code** or **binaries**
  - Instrument **statically** or **dynamically**
  - Instrument **input program** or **runtime system**
- Challenges when handling analysis:
  - **In-memory representation of measurements** (for compression or speed)
  - **Emitting measurements**

Perturbation

Example challenge: can we use total counts to decide *whether* to optimise some function `f`?

- On each method entry: get current time
- On each method exit: get time again, update aggregate
- Reading timer takes: ~ 80 cycles
- Short `f` calls may be much faster than 160 cycles
  - `fun f(x) = x + 1 // ca. 0.25 cycles`
  - `fun f(x) = x     // ca. 0 cycles`
- Also: measurement needs CPU registers
  $\Rightarrow$ may require registers
  $\Rightarrow$ may slow down code further

1 GHz CPU: 1 cycle $= 10^{-9}s$ (1 nanosecond / ns)

Measurements perturb our results, slow down execution

Sampling

Alternative to full counts: *Sampling*

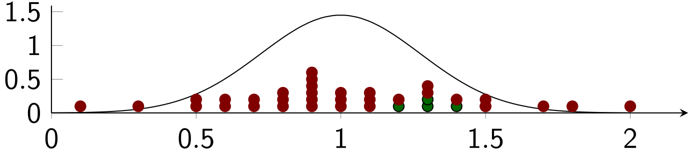
- Periodically interrupt program and measure
- Problem: how to pick the right period?

1. System events (e.g., GC trigger or 'safepoint')
   System events may bias results
2. Timer events: periodic intervals
   - May also bias results for periodic applications
   - Randomised intervals can avoid bias
   - Short intervals: perturbation, slowdown
   - Long intervals: imprecision

Samples and Measurements

Samples are *collections of measurements*

- Bigger samples:
  - Typically give more precise answers
  - May take longer to collect
- Challenge: representative sampling



Carefully choose what and how to sample

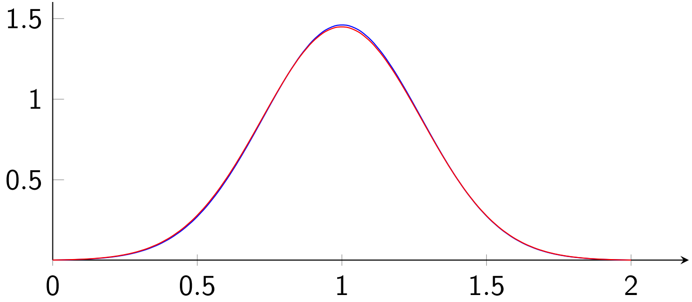
Summary

- We measure **Characteristics** of **Events**
- **Sample**: set of **Measurements** (of characteristics of events)
- Measurements often cause **perturbation**:
  - Measuring disturbs characteristics
  - Not relevant for all measurements
  - Measuring time: more relevant the smaller our time intervals get
- Can measure by:
  - **Counting**: observe every event
    - Gets all events
    - Maximum measurement perturbation
  - **Sampling**: periodically measure
    - Misses some events
    - Reduces perturbation

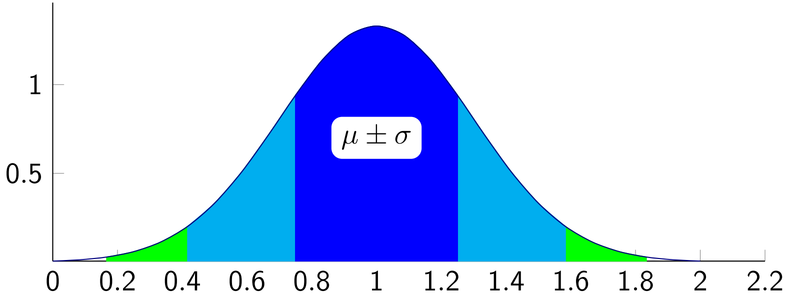
Presenting Measurements

	P1	P2
Mean μ	1,001	0,999
Standard Deviation σ	0,273	0,275

Assuming normal distribution:



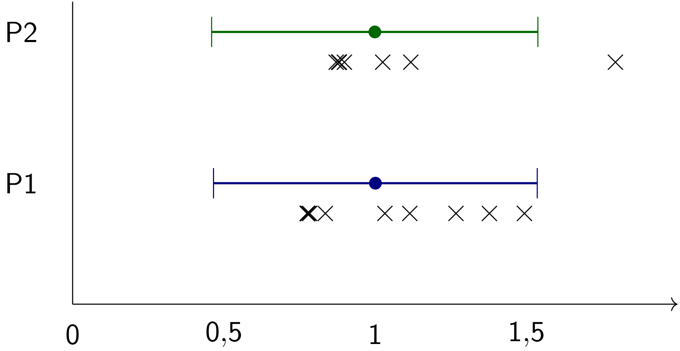
Standard Deviation, Assuming Normal Distribution



Deviation	**Chance of actual μ being in interval**
σ	68,27%
1,96σ	95,00%
2σ	95,45%
2,58σ	99,00%
3σ	99,73%

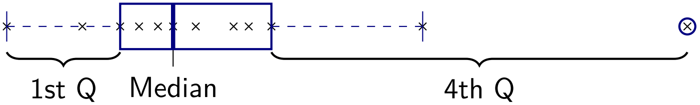
How Well Does Normal Distribution Fit?

Representation with error bars (95% confidence interval):



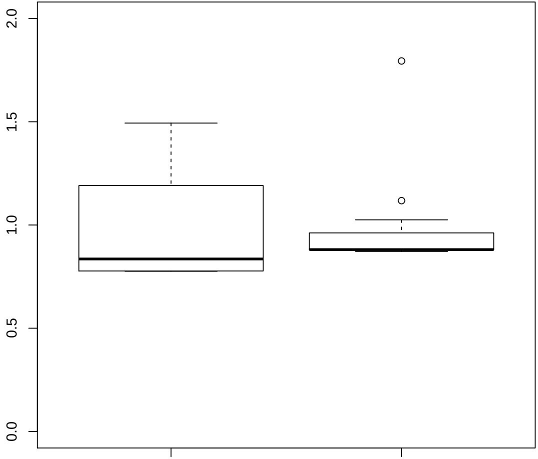
Mean + Std.Dev. are misleading if measurements don't observe normal distribution!

Box Plots

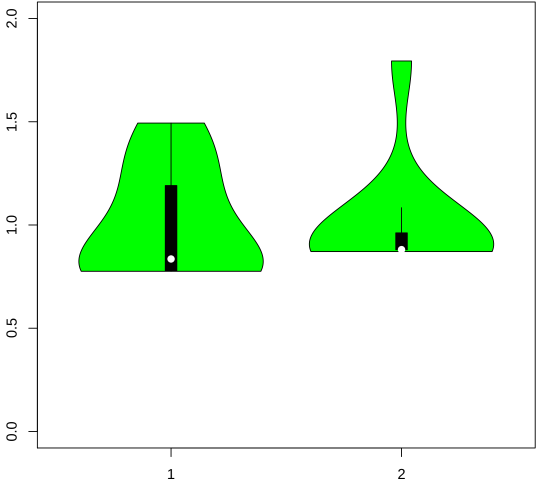


- Split data into 4 *Quartiles*:
  - Upper Quartile (1st Q): Largest 25% of measurements
  - Lower Quartile (4th Q): Smallest 25% of measurements
  - Median: measured value, middle of sorted list of measurements
- Box: Between 1st/4th quartile boundaries
  Box width = inter-quartile range (*IQR*)
- 1st Q whisker shows largest measured value $\leq 1{,}5 \times$ IQR (from box)
- 4th Q whister analogously
- Remaining *outliers* are marked

Box plot: example



Violin Plots



Summary

- We don't usually know our statistical distribution
- There exist statistical methods to work precisely with confidence intervals, given certain assumptions about the distribution (not covered here)
- Visualising without statistical analysis:
  - **Box Plot**
    - Splits data into **quartiles**
    - Highlights points of interest
    - No assumption about distribution
  - **Violin Plot**
    - Includes Box Plot data
    - Tries to approximate probability distribution function visually
    - Can help to identify actual distribution

Gathering Dynamic Data

- Instrumentation and Software Probes
  - Example: Performance profiler
- Simulation (or Emulation)
  - Example: CPU simulator
- Hardware Probes
  - Example: Hardware Performance Counters

Automatic Performance Measurement

- **[Software Probes]** Profiler:
  - Interrupts program during execution
  - Examines call stack
- **[Software Probes]** Operating System Perf. Counters:
  - Count important system events (network accesses etc.)
- **Simulator**:
  - Simulates CPU/Memory in software
  - Tries to replicate inner workings of machine
  - Alternatively: *Emulator* (= replicate only observable functionality, not internals)
- **[Hardware Probes]** CPU:
  - Hardware performance counters count interesting events

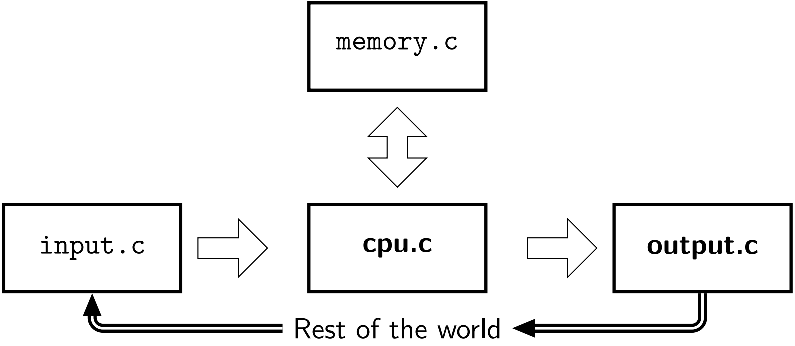
Profiler

- Measures: which functions are we spending our time in?
- Approach:
  - Build stack maps
  - Execute program, interrupt regularly
  - During interrupt:
    - Examine program counter
    - Examine stack
- Infer callers from stack contents

Execution Stack
`return` (old-1)
`$fp` (old-1)
...
...
`return` (old-2)
`$fp` (old-2)
...

Source of inaccuracy: inlined functions don't track their caller on call stack

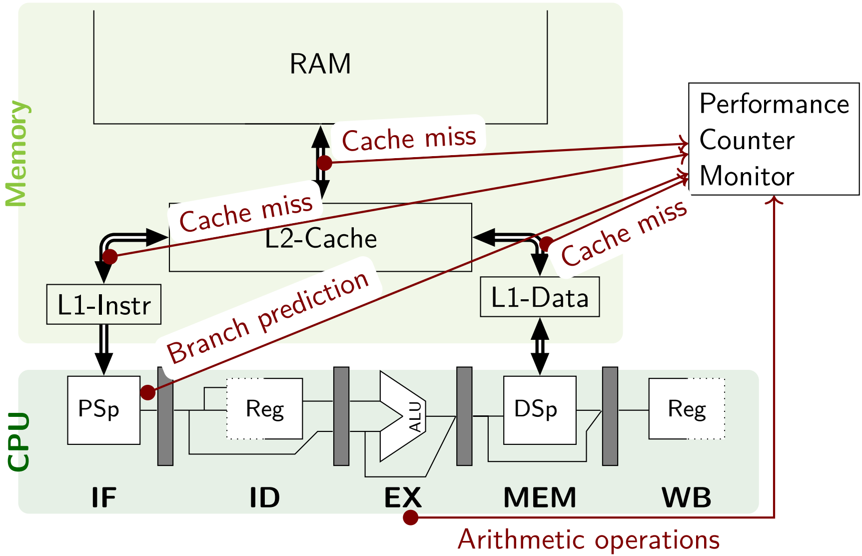
Simulator



- Software simulates hardware components
- Can count events of interest (memory accesses etc.)

Modern CPUs are very complex: Simulators are inaccurate in practice

Hardware Performance Counters (1/2)



Hardware Performance Counters (2/2)

Special CPU registers:

- Count *performance events*
- Registers must be configured to collect specific performance events
  - Number of CPU cycles
  - Number of instructions executed
  - Number of memory accesses

    ...
- #performance event types > #performance registers

May be inaccurate: not originally built for software developers

Summary

- Performance analysis may require detailed dynamic data
- **Profiler**: Probes stack contents at certain intervals
- **Simulator**:
  - Simulates hardware in software, measures
  - Tends to be inaccurate
- **Performance Counters**:
  - Software:
    - Operating System counts events of interest
  - Hardware:
    - Special registers can be configured to measure CPU-level events

Gathering Dynamic Data

- Instrumentation and Software Probes
- Simulation
- Hardware Probes

Generality of Performance Measurements?

Measured performance properties are valid for…

- Selected CPU
- Selected operating system
- Compiler version and configuration
- Operating system configuration:
  - OS setup
    (e.g., dynamic scheduler)
  - Processes running in parallel

    …
- A particular input/output setup
  - Behaviour of attached devices
  - Time of day, temperature, air pressure, …
- CPU configuration (CPU frequency etc.)

…

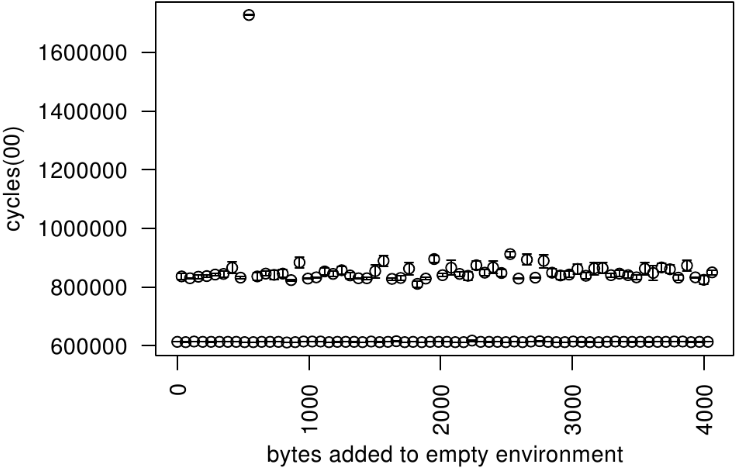
Is that all?

Unexpected Effects

- User `toddm` measures run time 0.6s
- User `amer` measures run time 0.8s
- Both measurements are stable
- Reason for discrepancy:
  - Before program start, Linux copies shell environment onto stack
  - Shell environment contains user name
  - Program is loaded into different memory addresses
    $\Rightarrow$ Memory caches can speed up memory access in one case but not the other

Changing your user name can speed up code

Unexpected Effects



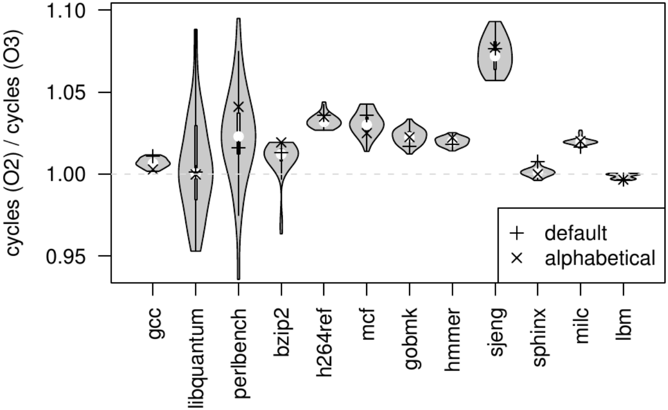
Mytkowicz, Diwan, Hauswirth, Sweeney: “Producing wrong data without doing anything obviously wrong”, in ASPLOS 2009

Linking Order

Is there a difference between re-ordering modules in RAM?

`gcc a.o b.o -o program` (Variant 1)

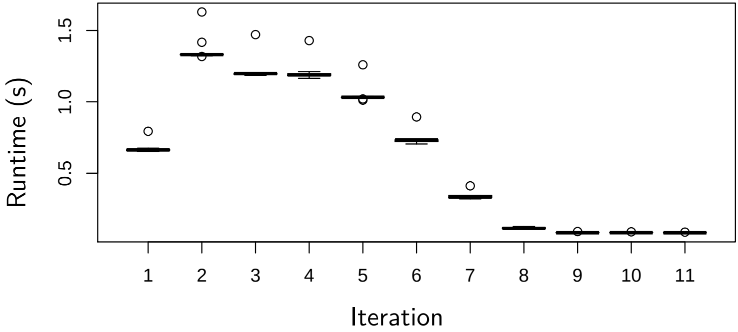
`gcc b.o a.o -o program` (Variant 2)



(Mytkowicz, Diwan, Hauswirth, Sweeney, ASPLOS'09)

Adaptive Systems

- Java program: loop *n* iterations (x axis) around simple computation that randomly samples from pre-initialised array
- Measurement: 11 runs
  - Ran each *n* 11 times, time reported below is last iteration only



Warm-up effect

Warm-Up Effects

- Performance varies during initial runs
- Eventually reaches steady state
- Reason: Adaptive Systems
  - Hardware:
    - *Cache*: Speed up some memory accesses
    - *Branch Prediction*: Speed up some jumps
    - *Translation Lookaside Buffer*
  - Software:
    - *Operating System / Page Table*
    - *Operating System / Scheduler*
    - *Just-in-Time compiler*
- Understanding performance: what to measure?
  - Latency: measure first run
    Reset system before every run
  - Throughput: later runs
    Discard initial n measurements

Ignored Parameters

- Performance affected by subtle effects
- Systems developers must “think like researchers” to spot potential influences

Beware of generalising measurement results!

Summary

- Modern computers are complex:
  - *Caches* make memory access times hard to predict
  - *Multi-tasking* may cause sudden interruptions
  - *CPU frequency scaling* changes speed based on temperature

  ...
- This makes measurements difficult:
  - Must carefully consider what **assumptions** we are making
  - Must measure repeatedly to gather **distribution**
  - Must check for **warm-up effects**
  - Must try to understand causes for performance changes
- Measurements are often not normally distributed
  - Mean + Standard Deviation may not describe samples well
  - If in doubt, use **box plots** or *violin plots*

Summary: Dynamic Analysis

- Collecting *Measurements* of *Characteristics* at *Events* via *Probes*:
  - In software, hardware, or indirectly via simulation
- Applications include:
  - Purely to observe (program understanding etc.)
  - Efficiency (JIT compilation etc.)
  - Prevent undesirable behaviour (Safety, Security)
- *Sampling* to reduce overhead:
  - Finite set of inputs/workloads, hardware etc.
- Some characteristics (esp. *performance*) influenced by *sources of variability* outside of program and program input
- Can usually avoid false positives, can*not* usually avoid false negatives

Outlook

- Oral exam information on Thursday
- Oral exam registration on Friday
- Final Lecture: (Mostly) review session– bring your questions!

http://cs.lth.se/edap15