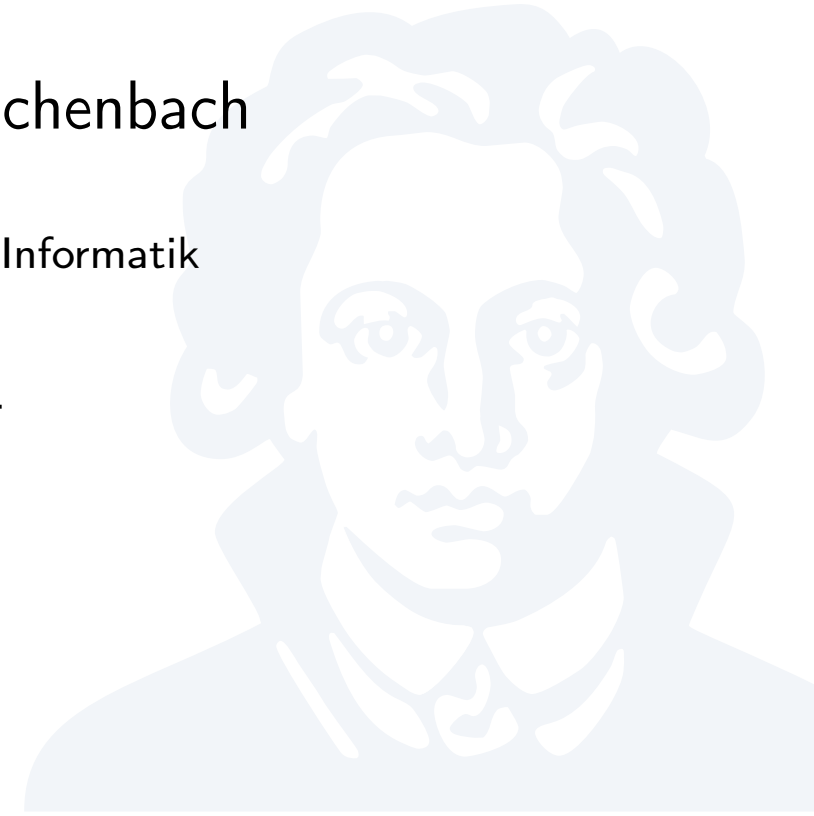
Software Engineering Tools 03
Dynamic Program Analysis

Prof. Dr. Christoph Reichenbach

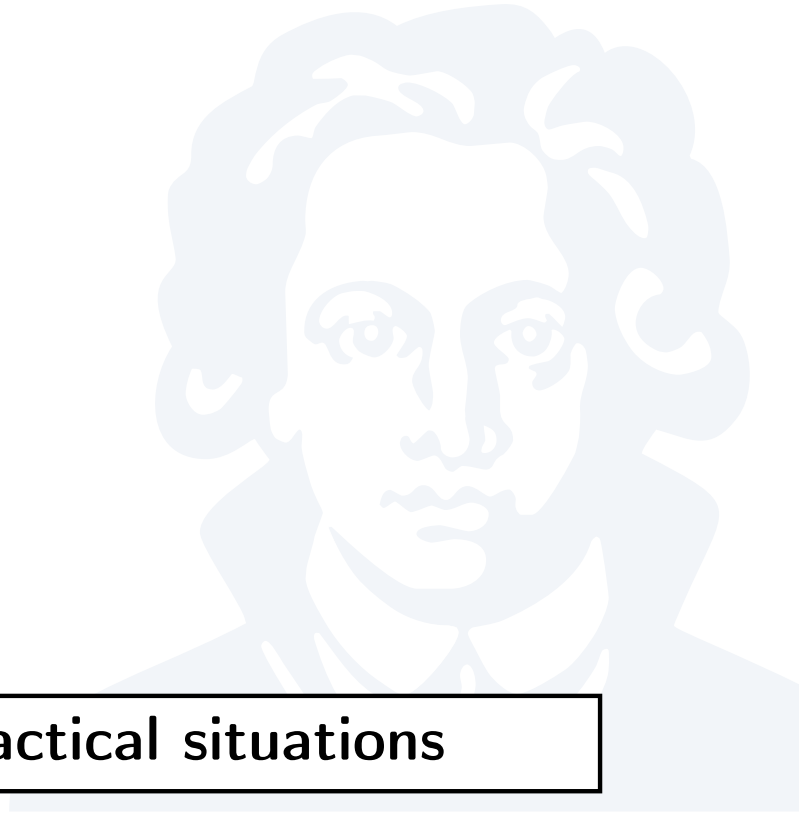
Fachbereich 12 / Institut für Informatik

30. April 2014



The Limits of Static Analysis

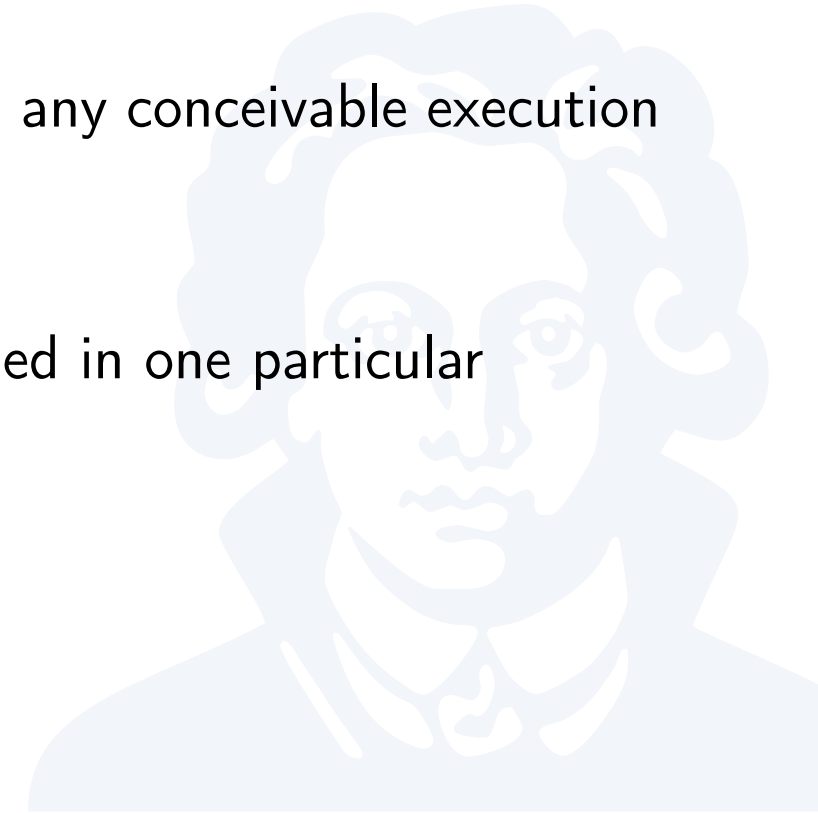
- How long does it take to execute this piece of code?
- How important is this piece of code in practice?
- How well does this code collaborate with hardware devices?
  - Caches?
  - Branch predictors?
  - The ALU(s)?
  - The TLB?
  - Harddisks?
  - Networking devices?
  - ...



Impossible to predict for all practical situations

Static and Dynamic analysis

- Static analysis:
  - Analysis predicts execution
  - Captures ‘what could happen in any conceivable execution
- Dynamic analysis:
  - Execution informs analysis
  - Captures ‘what actually happened in one particular execution



A simple example

```
void frob(int[] data, int len)
{
  for (int i=0; i < len; i++) {
    data[len + i] = data[i];
    data[i] *= data[i];
} }

void f(...) {
  int d = new int[6];
  ...
  frob(d, 4);
}

void g(...) {
  int d = new int[42000];
  ...
  frob(d, 21000);
}
```

Possible optimisations

- *Loop unrolling*: duplicate loop body to remove jump instructions
- *Vectorisation*: merge adjacent arithmetic operations into one
- Inlining
- *Bounds check elimination*: rem. array-out-of-bounds tests

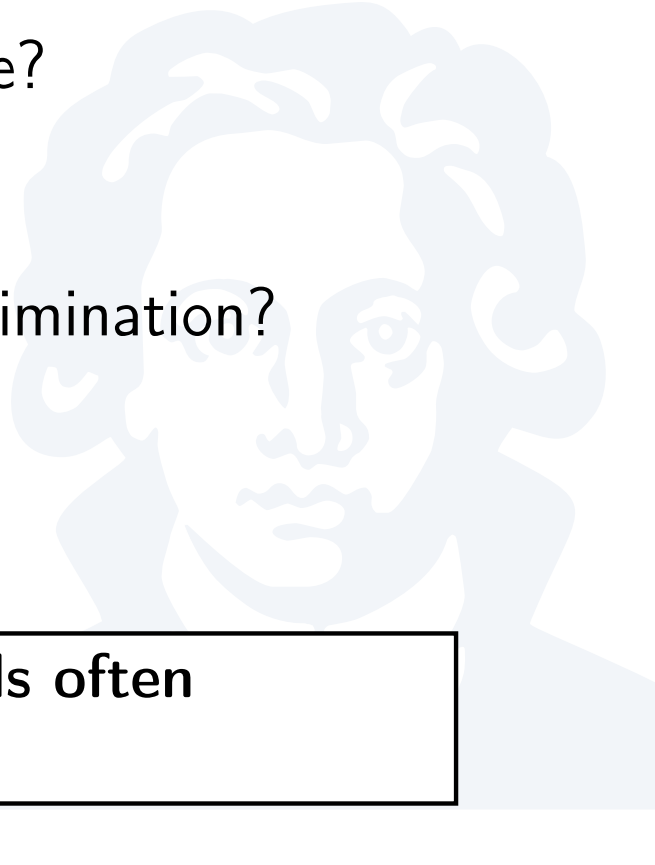
. . . but are they worth it?

Context-sensitive static analysis *may* help, but there will always be cases where it can't.

Profile-Driven Optimisation

Idea: *Measure run-time properties* to guide optimisation

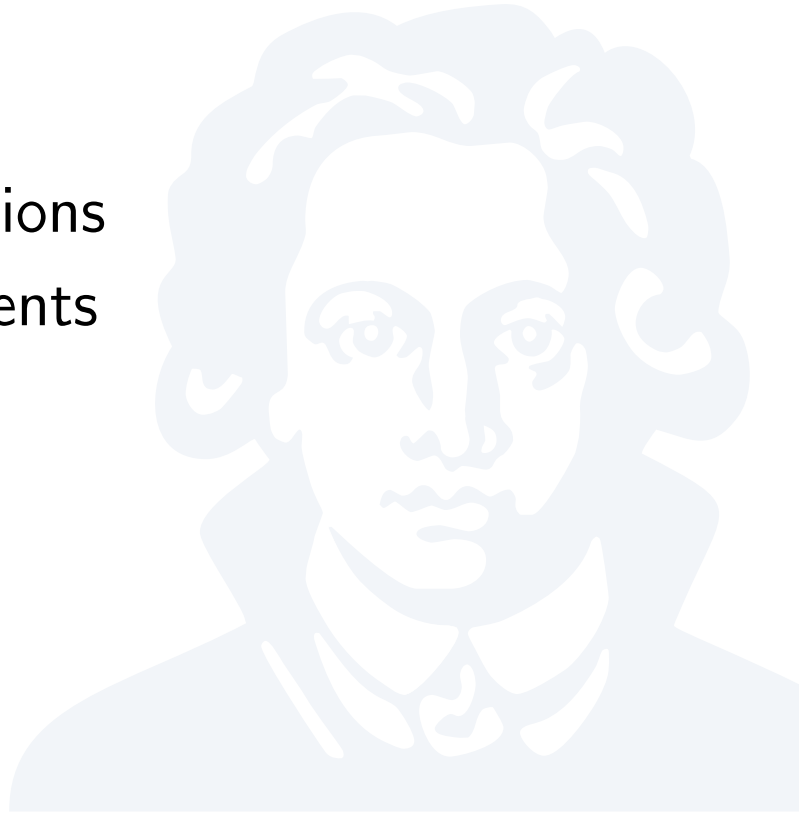
- Loop unrolling
  Does 'f' get called with large arrays?
- Vectorisation
  Is substantial loop unrolling taking place?
- Inlining
  Does 'f' get called with small arrays?
  Or would this facilitate bounds check elimination?
- Bounds check elimination
  Are we inlining 'f'?



No hard requirements, thresholds often machine-depenent

Concepts

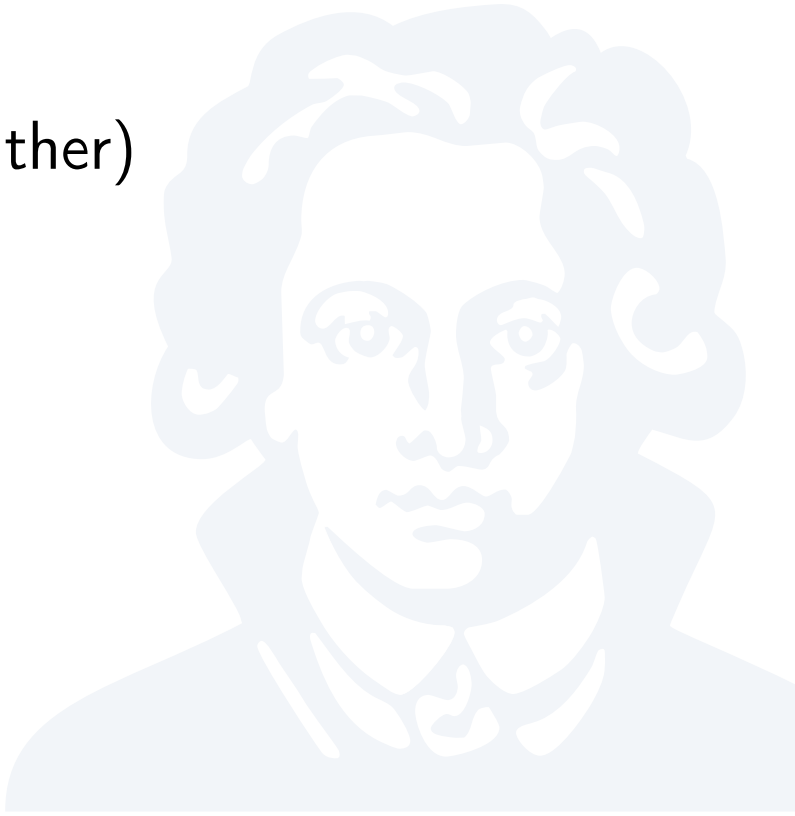
- *Events*: When we measure
- *Characteristics*: What we measure
- *Measurements*: Individual observations
- *Samples*: Collections of measurements



Events

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

...

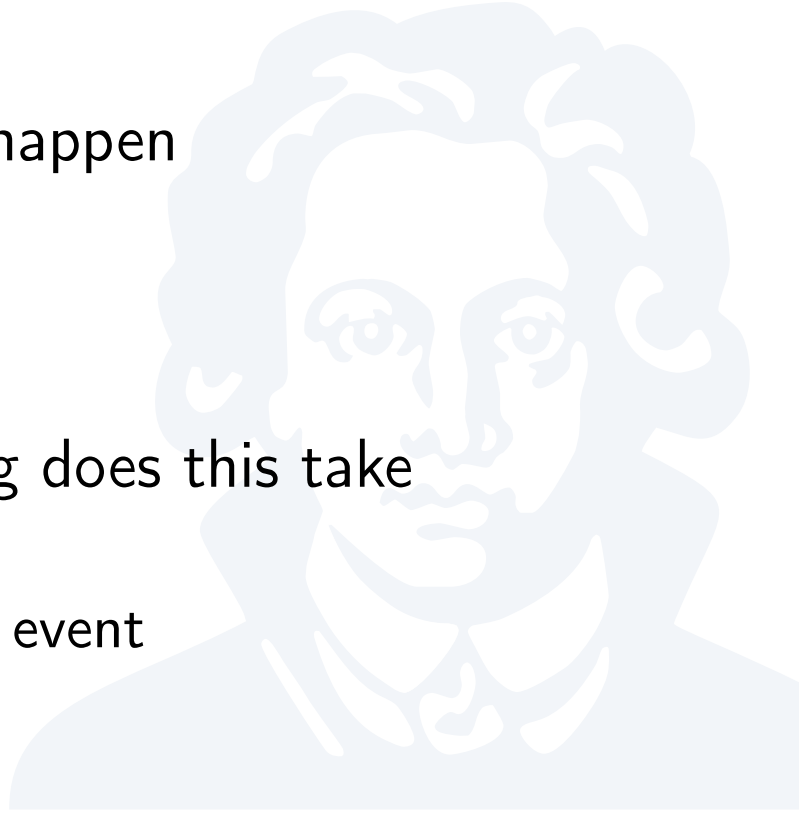


Characteristics

- *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
  - Compared

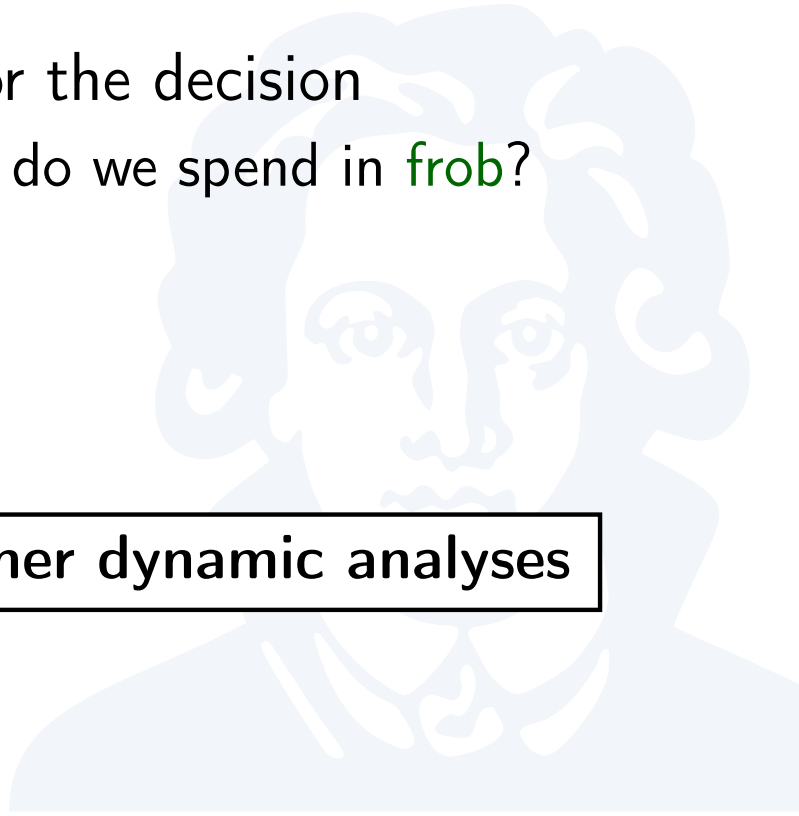


Running Example

JIT: should we opt-compile frob?

Decision requires

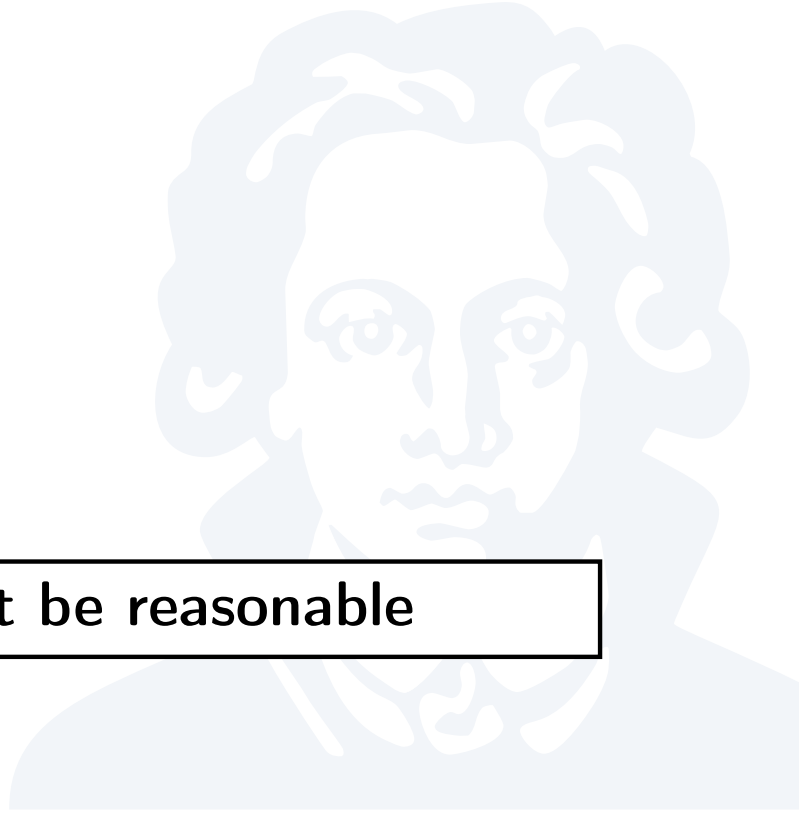
- *Policy*: how do we decide?
- *Analysis*: collect all data needed for the decision
  - What amount of execution time do we spend in frob? (How '*hot*' is it?)



Policy may also be derived from other dynamic analyses

How to Measure

- Run-time methodology:
  - Total counts
  - Sampling
- Analysis Architecture:
  - Instrumentation
  - Emulation and Debug Execution
  - Performance Counters

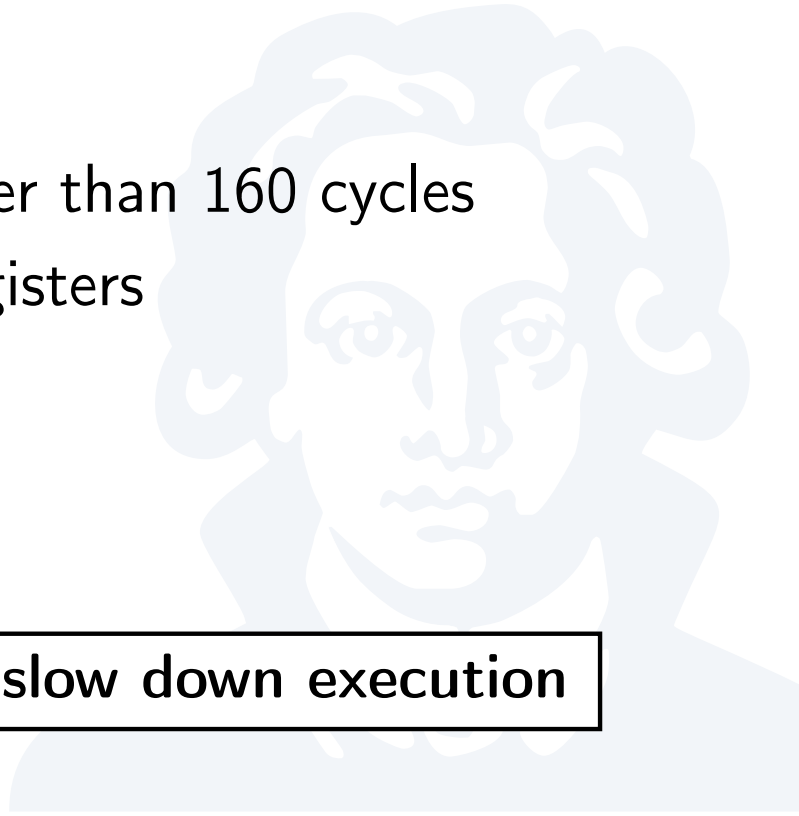


Each of these may or may not be reasonable

Perturbation

Can we use total counts to decide whether to optimise frob?

- On each method entry: get current time
- On each method exit: get current time again, update aggregate
- Reading timer takes: $\sim$ 80 cycles
- Short frob calls may be much faster than 160 cycles
- Also: measurement needs CPU registers
  $\Rightarrow$ may cause register spilling
  $\Rightarrow$ may slow down code further

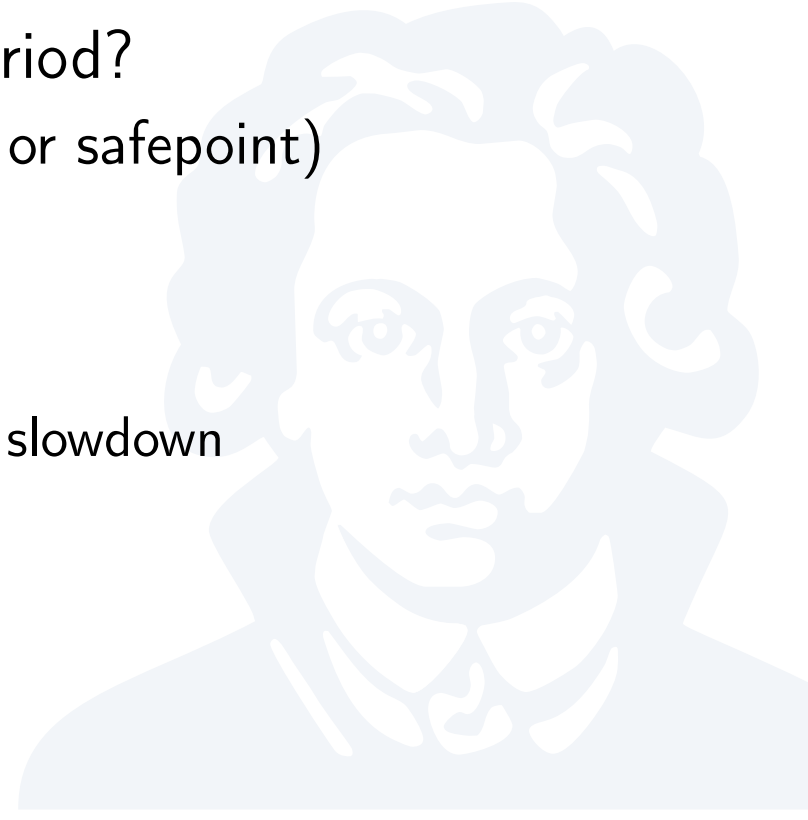


Measurements perturb our results, slow down execution

Sampling

Alternative: *Sampling*

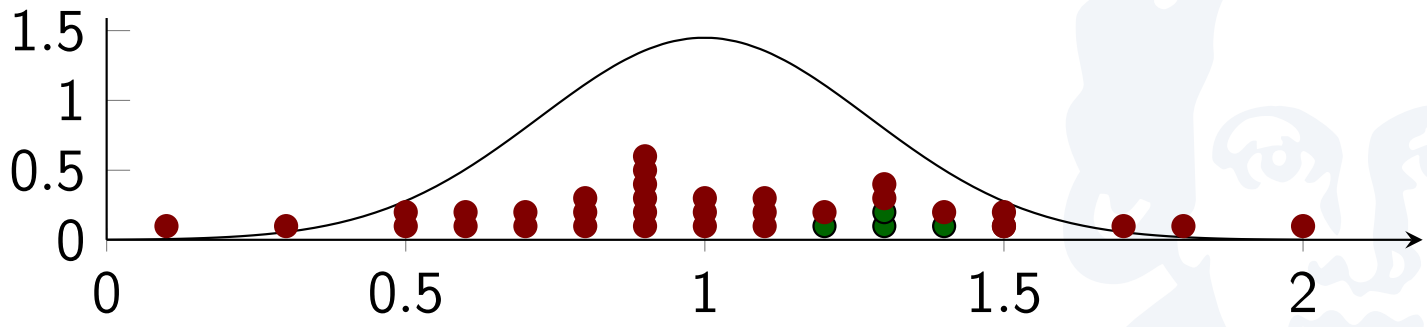
- Periodically interrupt program and measure
- Problem: how to pick the right period?
  1. System events (e.g., GC trigger or safepoint)
     System events may bias results
  2. Timer events: periodic intervals
     not biased
     - Short intervals: perturbation, slowdown
     - Long intervals: imprecision



Samples and Measurements

Samples are *collections of measurements*

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

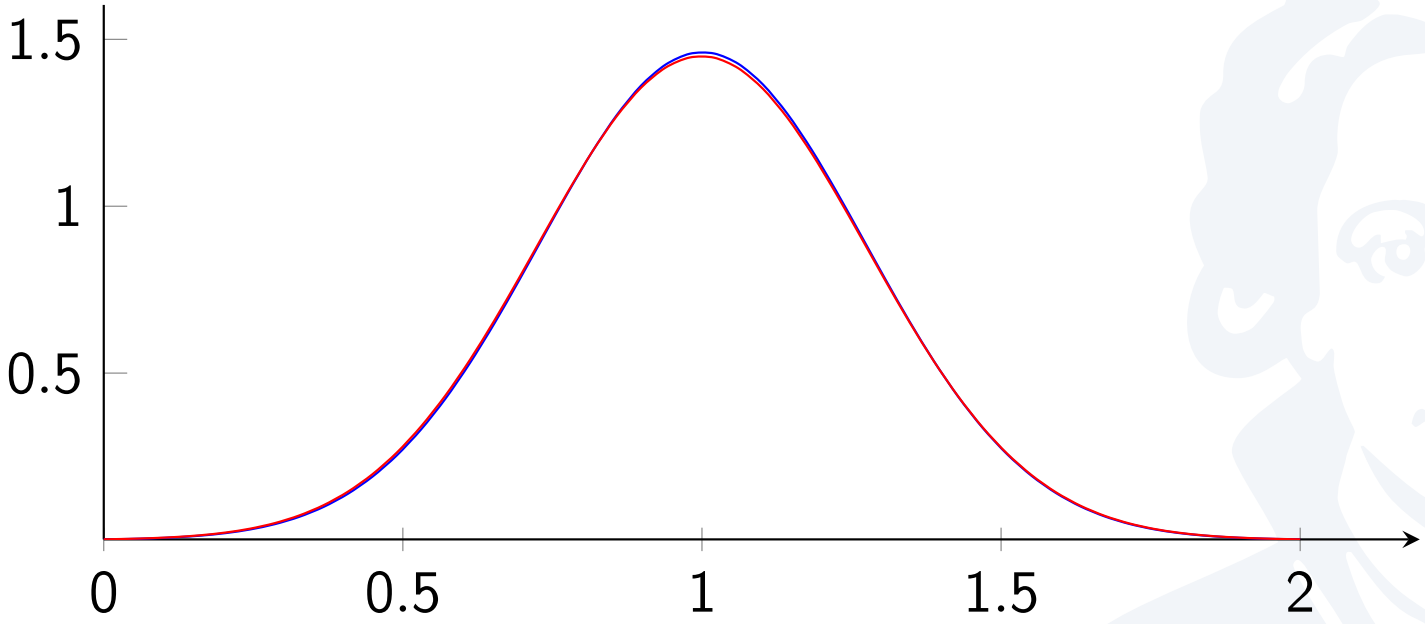


Carefully choose what and how to sample

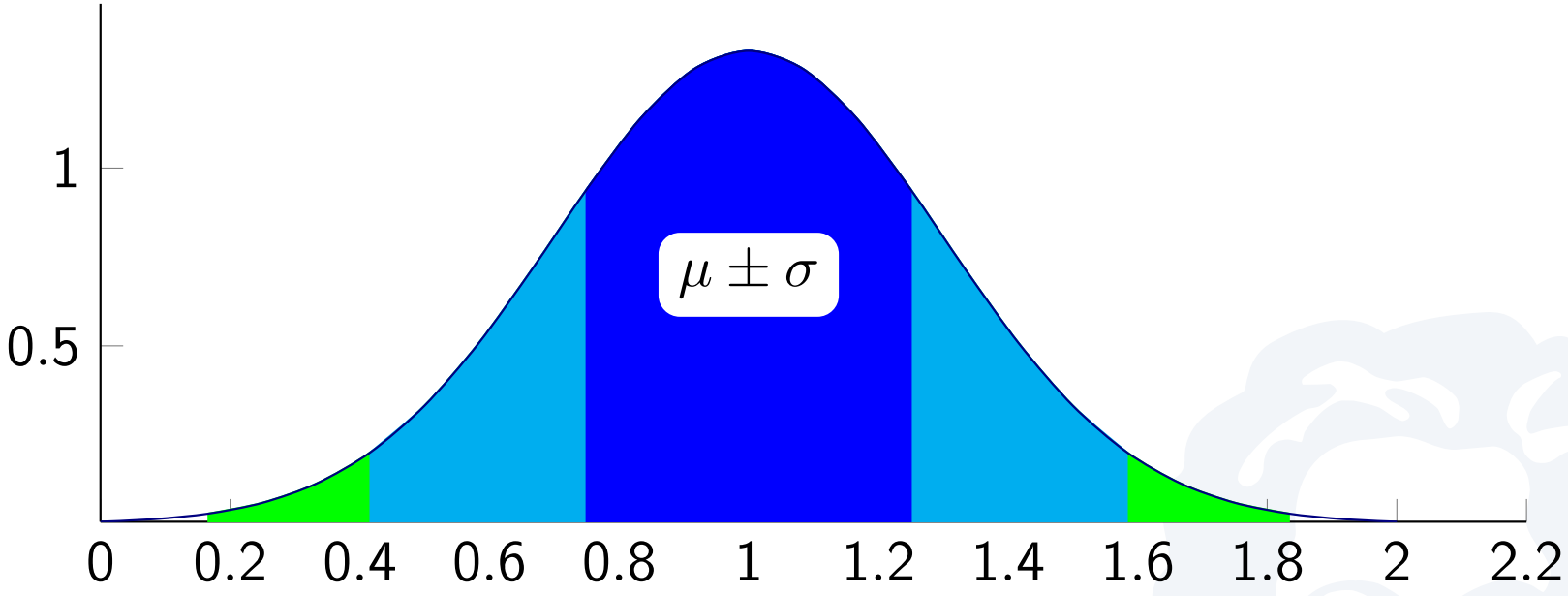
Presenting Measurements

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

Assuming normal distribution:



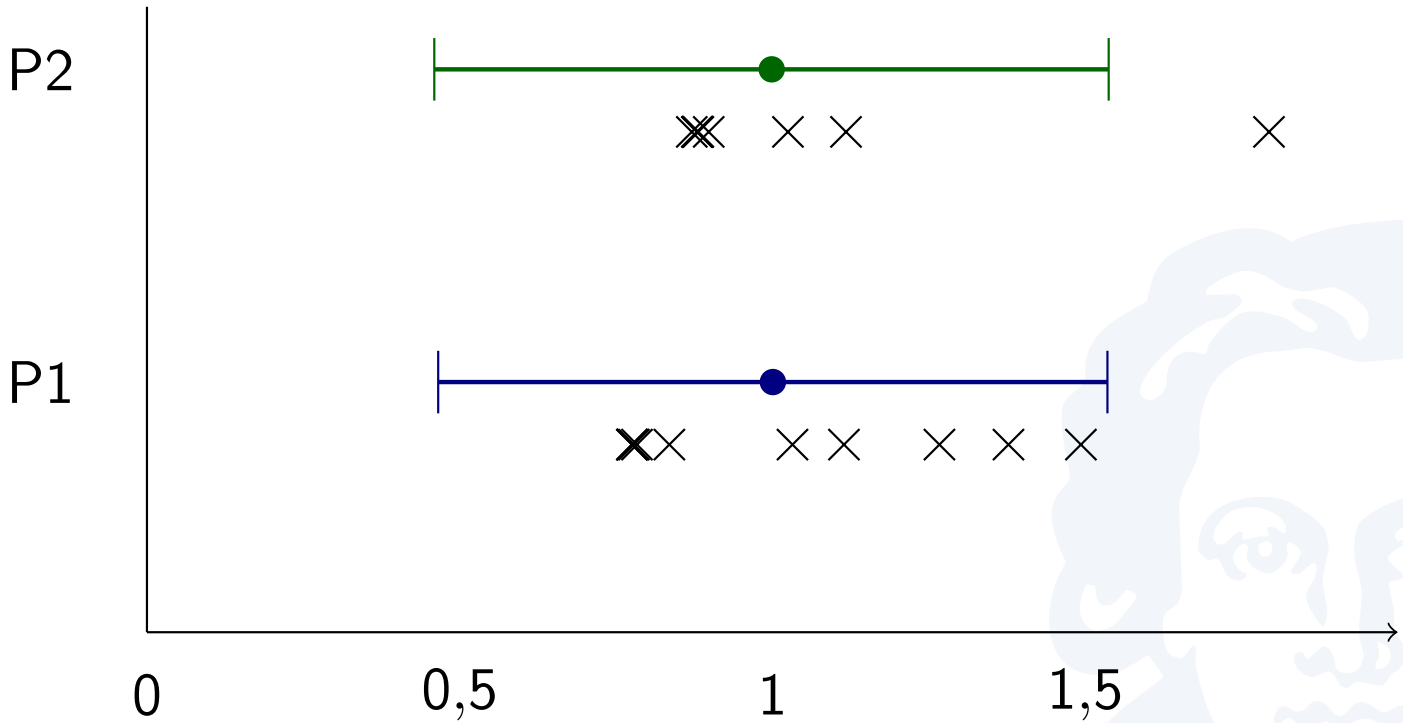
Standard Deviation, assuming Normal Distribution



Deviation	Number of measurements included
σ	68,27%
$1{,}96\sigma$	95,00%
2σ	95,45%
$2{,}58\sigma$	99,00%
3σ	99,73%

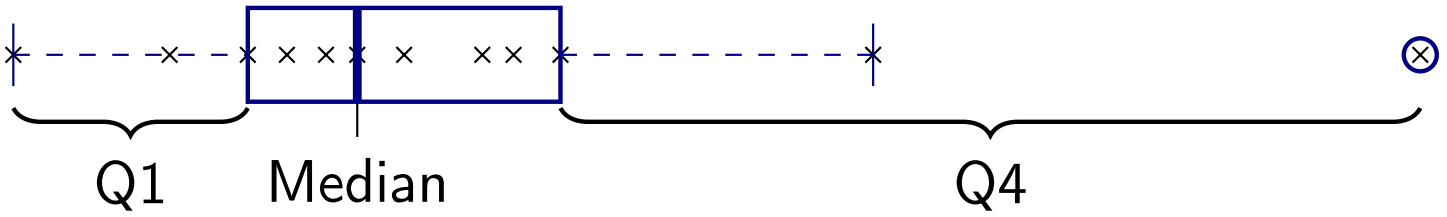
Matching Normal Distribution

Presentation with error bars (95% confidence interval):



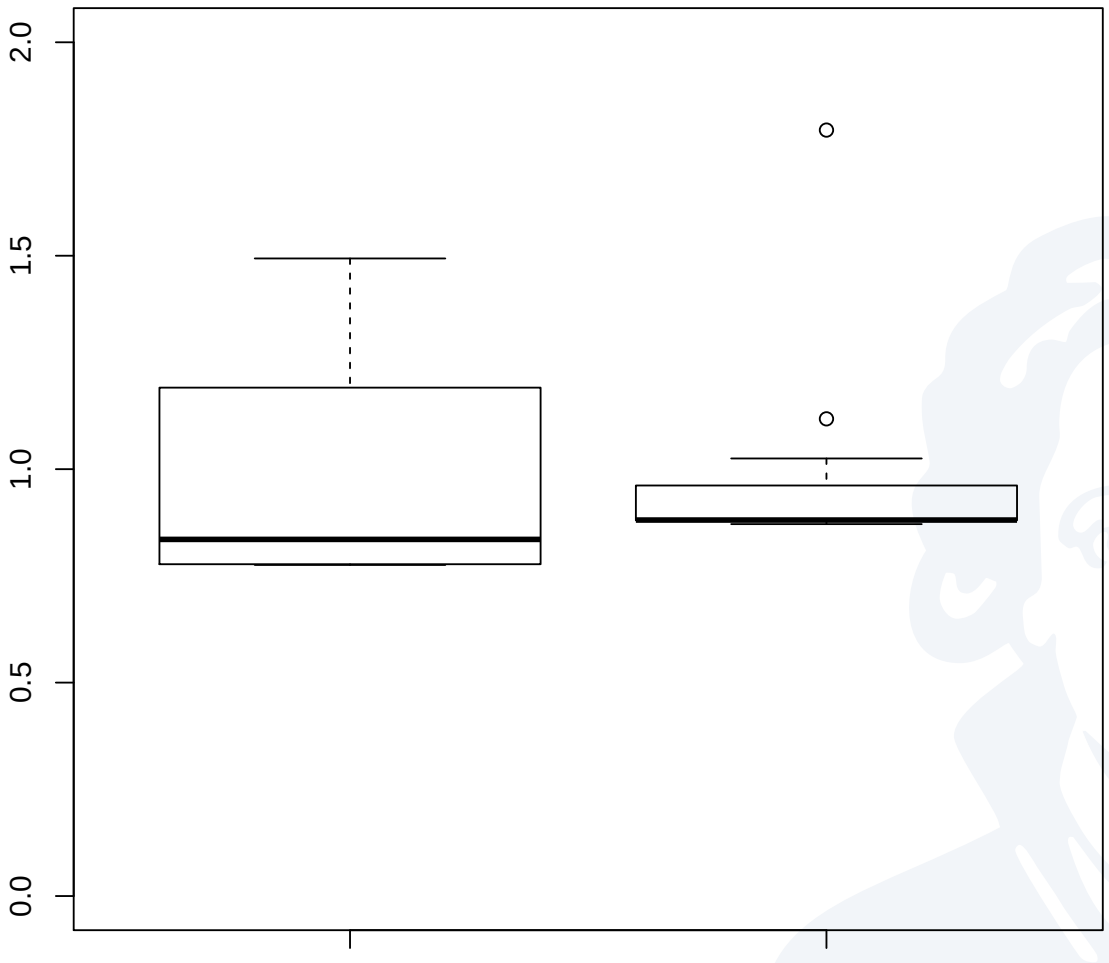
Mean and standard deviation may be misleading if measurements don't follow a normal distribution

Tukey Box Plots

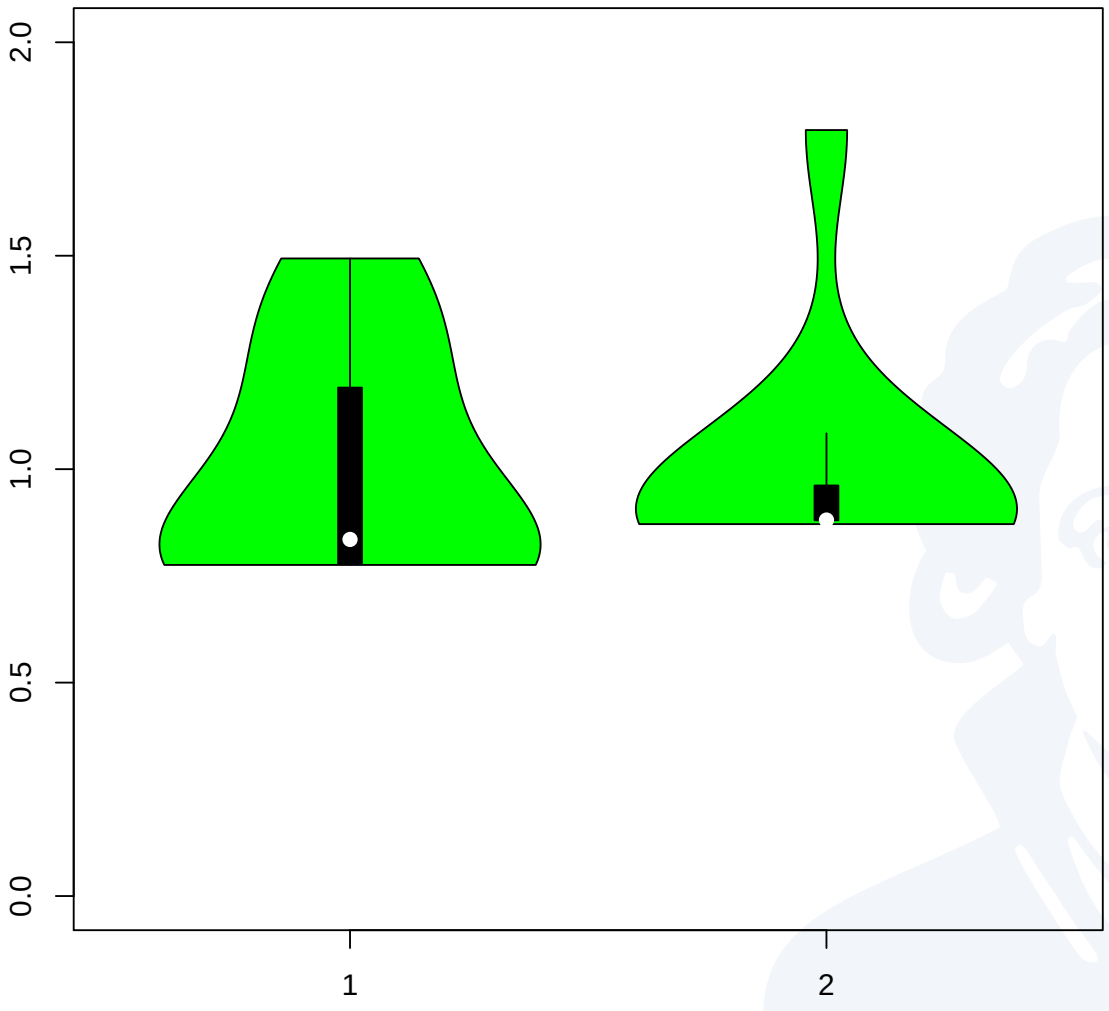


- Data split into 4 *quartile ranges*:
  - Upper quartile (Q1): the smallest 25% of our measurements
  - Lower quartile (Q4): the largest 25% of our measurements
  - Median separates Q2 and Q3 quartiles
- Box: between Q1 and Q4 boundaries
  Box width = *interquartile range* (IQR)
- *Whisker* in Q1 shows largest measured value $\leq 1{,}5 \times$ IQR (starting at box)
- Q4 whisker analogously
- Remaining *outliers* marked explicitly

Box Plot: Example

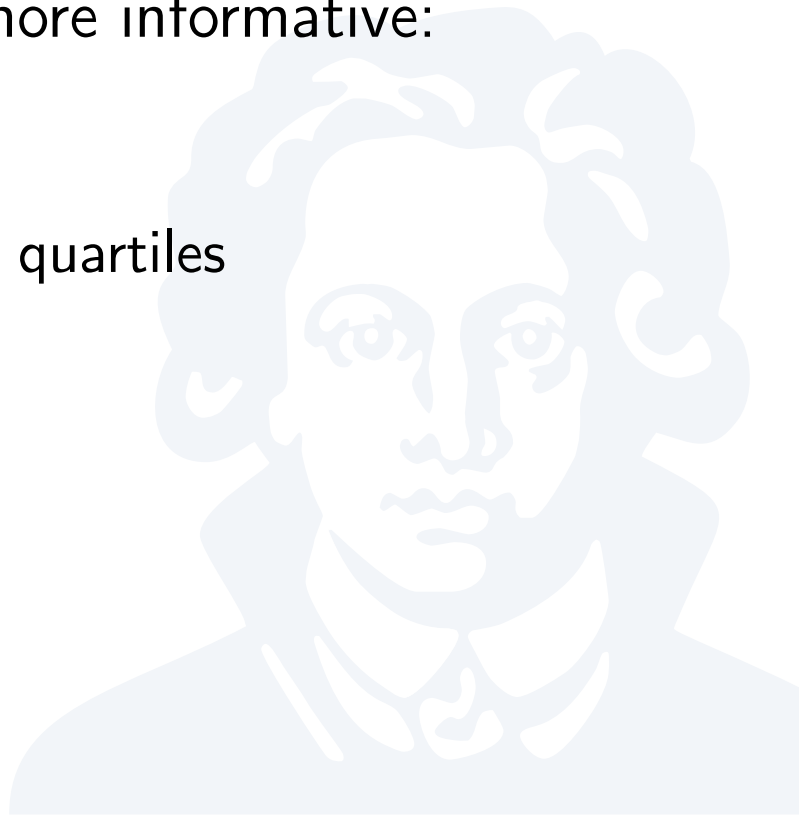


Violin Plot



Summary: Presenting Measurements

- Arithmetic mean, standard deviation only representative for normally distributed data
- Otherwise, the following is often more informative:
  - Middle 50%
  - Median
  - Extrem values from upper/lower quartiles
  - Outliers
- Compact representation:
  - Box plots
  - Violin plots

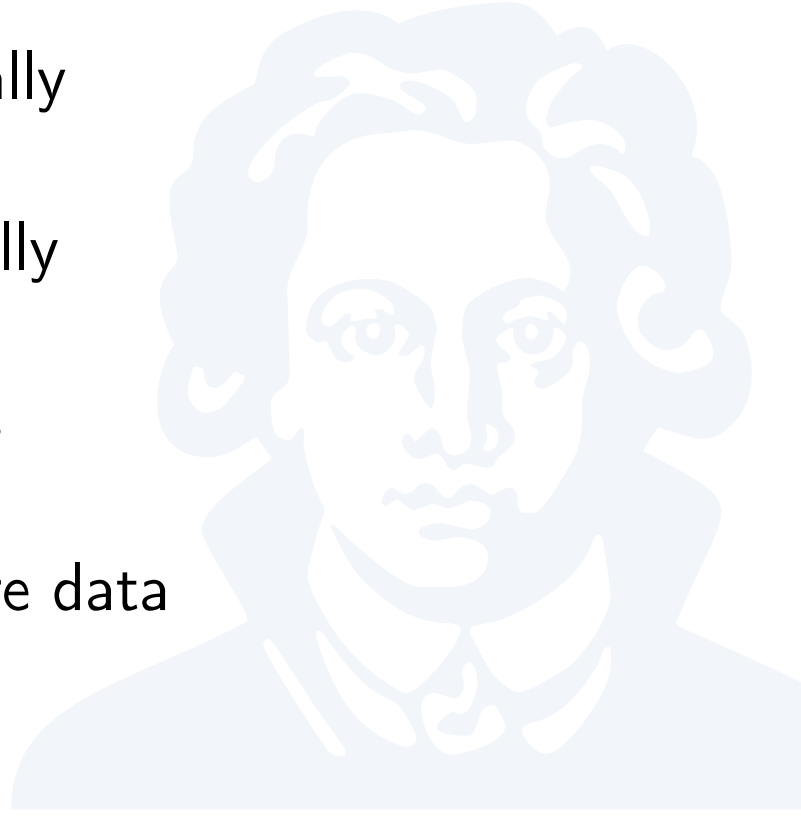


Analysis Architectures

- Instrumentation
  - Piggy-backing
- Debug-evaluation and emulation
- Performance Counters
  - kernel
  - hardware

Instrumentation

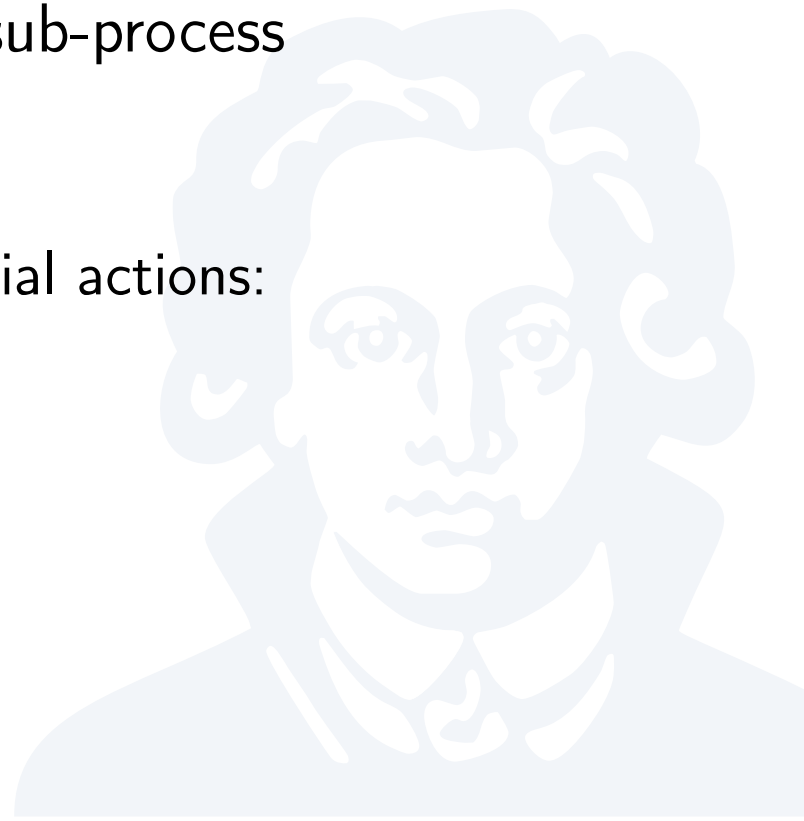
- *Manual*:
  Edit source code by hand
- *Automated source-to-source*:
  Transform source code automatically
- *Automated binary*: (pin etc.)
  Transform binary code automatically
- *Hooks*:
  Run-time system invokes callbacks
- *Piggy-backing*:
  Rewrite run-time system to capture data



Debug evaluation (ptrace)

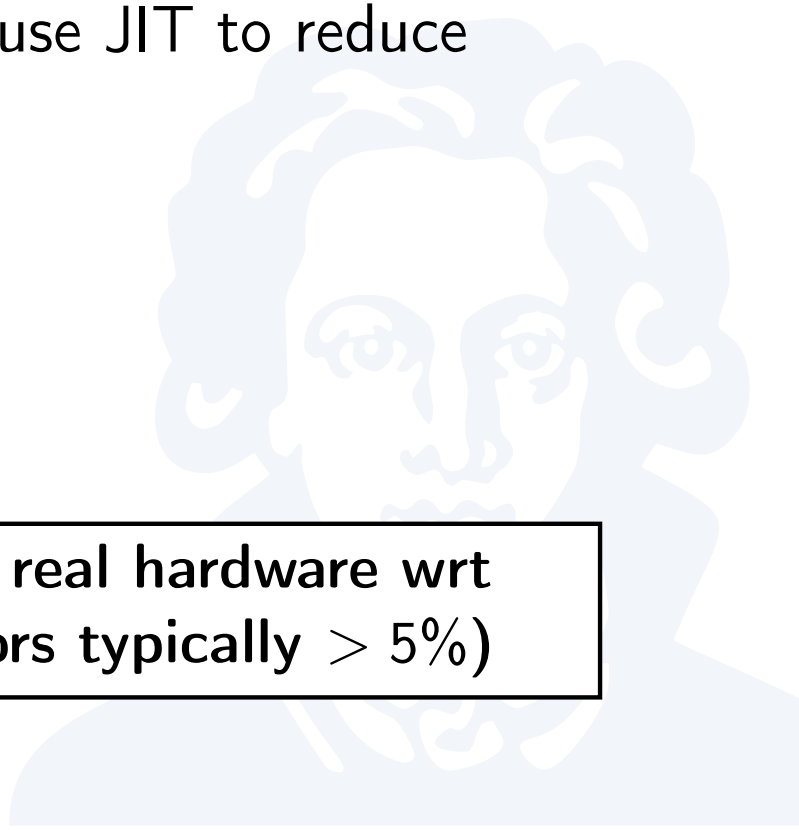
CPU/OS feature:

- *Debug process* starts program as sub-process
- Debug process may:
  - Execute program step-by-step
  - Run normally but intercept special actions:
    - System calls
    - Memory accesses



Emulators

- Simulate CPU, cache, TLB, other hardware
- Decode, execute machine instructions in software
- Current systems (valgrind, qemu) use JIT to reduce slowdown
- Highly versatile
- High overhead

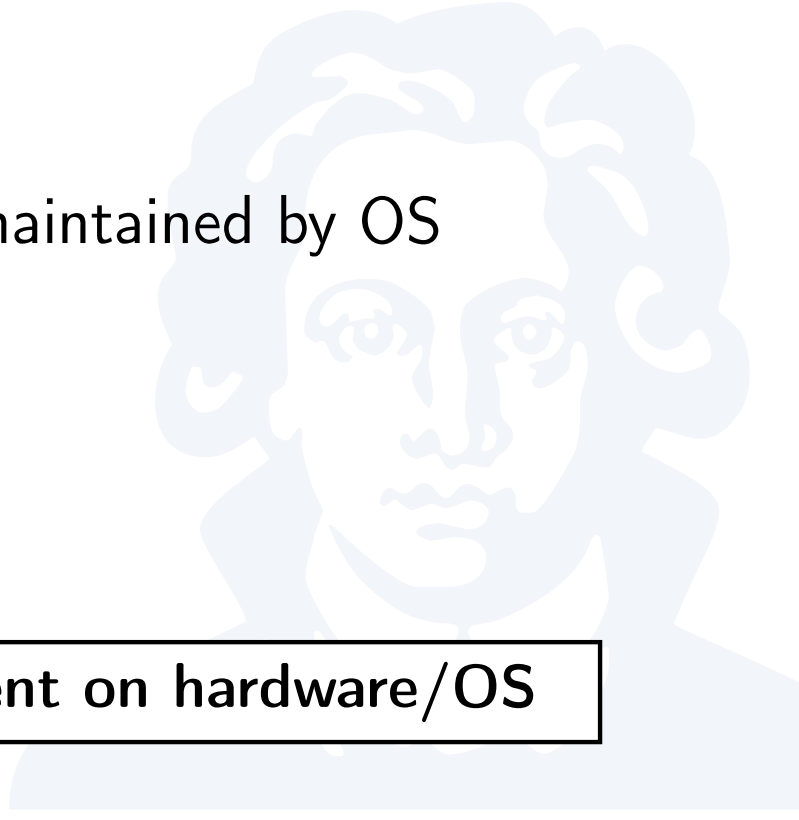


Emulators tend to disagree with real hardware wrt performance characteristics (errors typically $> 5\%$)

Performance Counters

Performance counters count events 'for free'

- Hardware Performance Counters: built into CPUs, controllers
  - # cache misses
  - # instructions executed
  - ...
- Software Performance Counters: maintained by OS
  - # page faults
  - # context switches
  - ...



Available statistics highly dependent on hardware/OS

PAPI

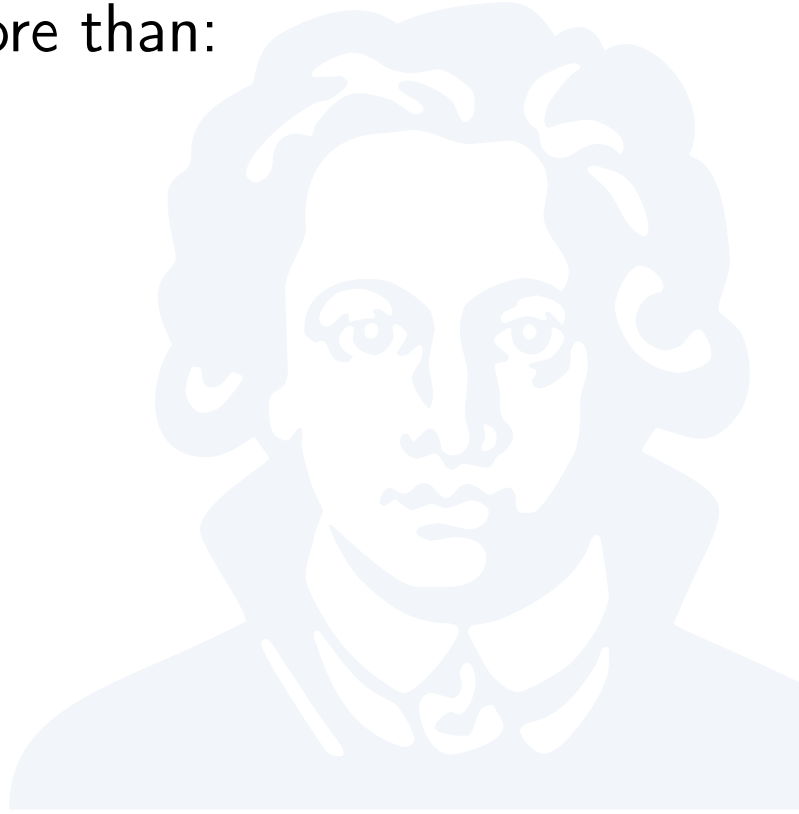
PAPI: easy-to-use performance counter API for Linux

```
// PAPI High-level API
const int PAPI_EVNR = 3;
int papi_events [PAPI_EVNR] = {
    PAPI_L1_DCM, // Level 1 data cache miss
    PAPI_L2_DCM, // Level 2 data cache miss
    PAPI_L3_DCM  // Level 3 data cache miss
};
long long papi_results[PAPI_EVNR];
if (PAPI_start_counters (papi_events , PAPI_EVNR)
    != PAPI_OK) error ();
run_benchmark(); // measure benchmark
if (PAPI_stop_counters (papi_results , PAPI_EVNR)
    != PAPI_OK) error ();
```

Be aware: PAPI calls tend to consume $>$ 10,000 cycles

Software Performance Counters

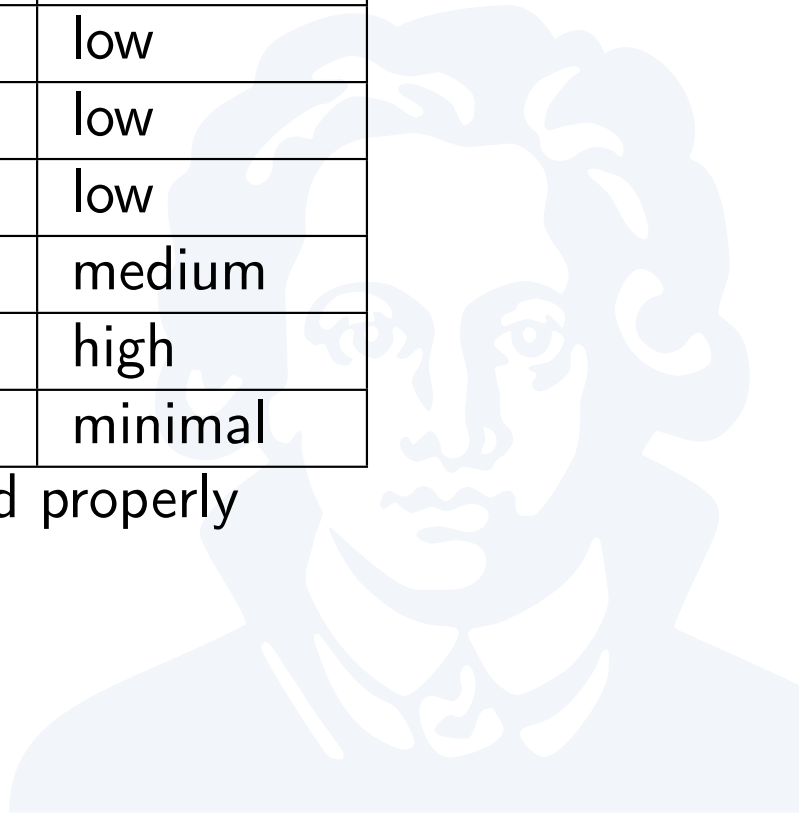
- Widely supported in modern OSes
- UNIX: `getrusage()`
- Capture OS-level events, rarely more than:
  - Memory usage
  - Time usage
  - Context switches
  - Blocks read/written
  - Page faults
  - Context switches
  - Interrupts/Signals



Summary: Analysis Architectures

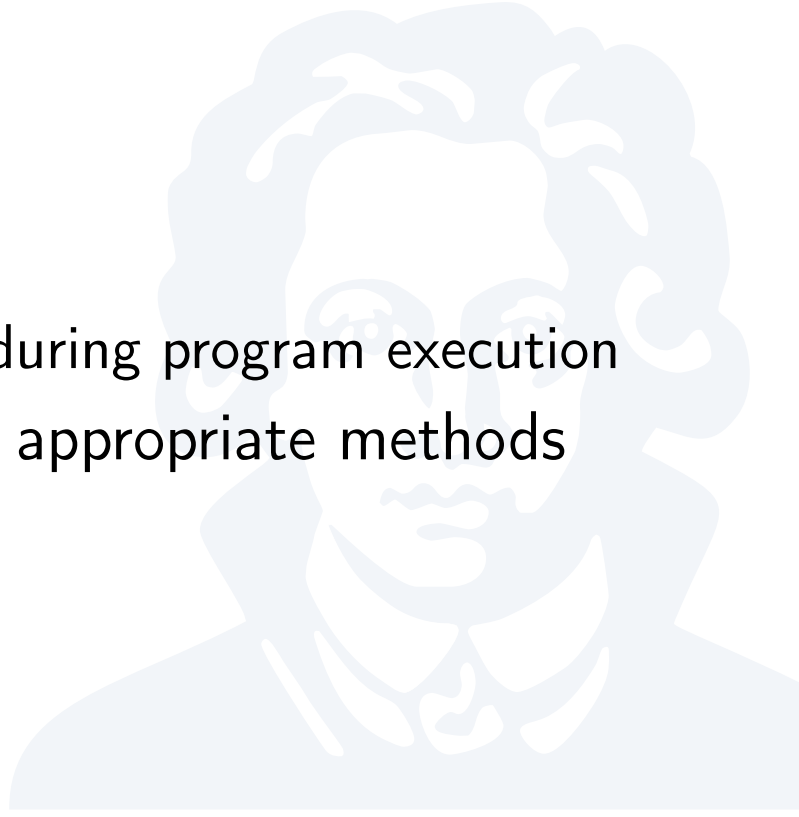
	Complete?	**Uses**	**Overhead**
Source-to-source	no	any	low
Binary instr.	no	any	low
Hooks	varies	limited	low
Piggy-backing	varies	limited	low
Debug mode	yes	limited	medium
Emulators	yes	any	high
Perfcounters	yes(?)	limited	minimal

Complete: will not miss any events, if used properly



Summary: How to Measure

- Choose events, characteristics
- Choose measurement methodology:
  - Total counts
  - Samples
- Pick suitable analysis architecture
- Collect data
  - May need to write out samples during program execution
- Analyse/compare with statistically appropriate methods
  - Box plots
  - Violin plots



Examples

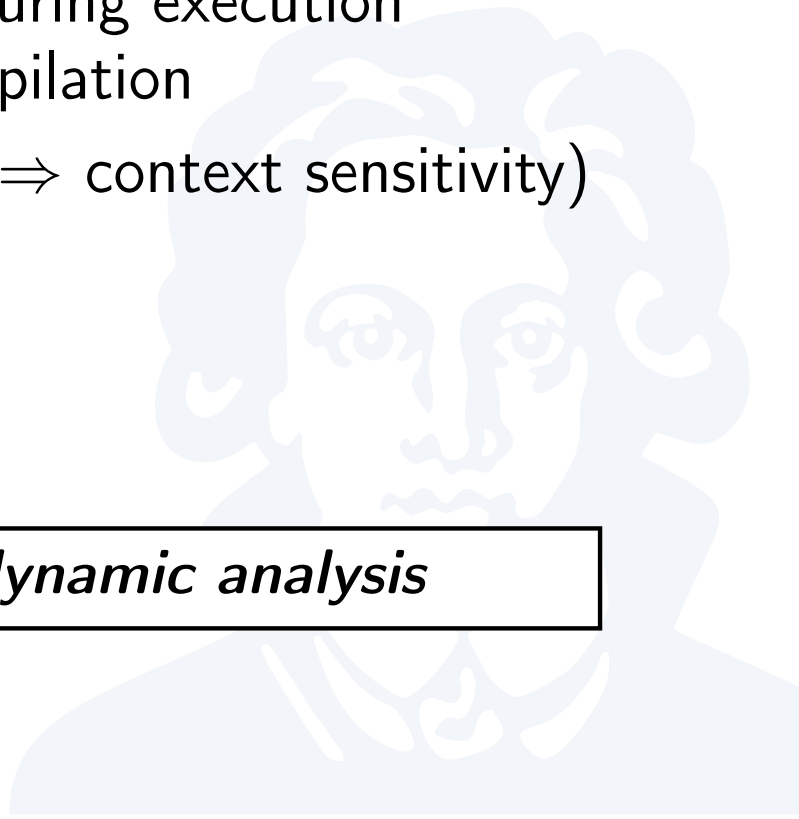
- Profile-Driven Optimisation
- Just-In-Time compilation
- Residual Analysis

Profile-Driven Optimisation

Recall our 'frob' example:

- gcc can use profiles for inlining and unrolling
- Collect array sizes, usage counts during execution
  Feed back into compiler for recompilation
- Inlining enables bounds checking ($\Rightarrow$ context sensitivity)
- Unrolling enables vectorisation

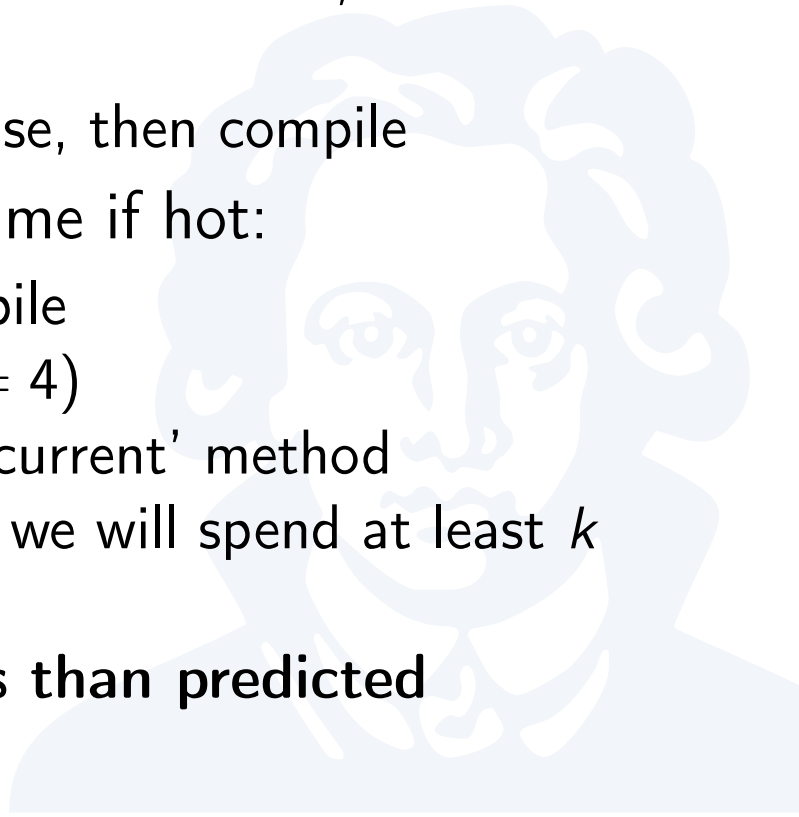
Example of *combined static/dynamic analysis*



Just-In-Time Compilation

The Jikes RVM JIT for Java:

- Efficient Java run-time system
- Two compilation modes:
  1. Baseline-compile: translate into machine code, zero optimisation
  2. Opt-compile: analyse and optimise, then compile
- Code may be *re-compiled* at run-time if hot:
  - Measure length of baseline-compile
  - Sample every n ms (default: $n = 4$)
  - Assume that we spent n ms in 'current' method
  - Assume: if we spent k ms total, we will spend at least k additional ms
  - **Is predicted compile time less than predicted execution time gain?**



Residual Analysis

FindBugs bug checker:

- Uses static analysis to detect 'likely' bugs in Java code
- Example: class C overrides equals() but not hashCode()
- *probably* behaves incorrectly in HashMap/HashSet

But what if C is never used with HashMap/HashSet?

Residual Analysis:

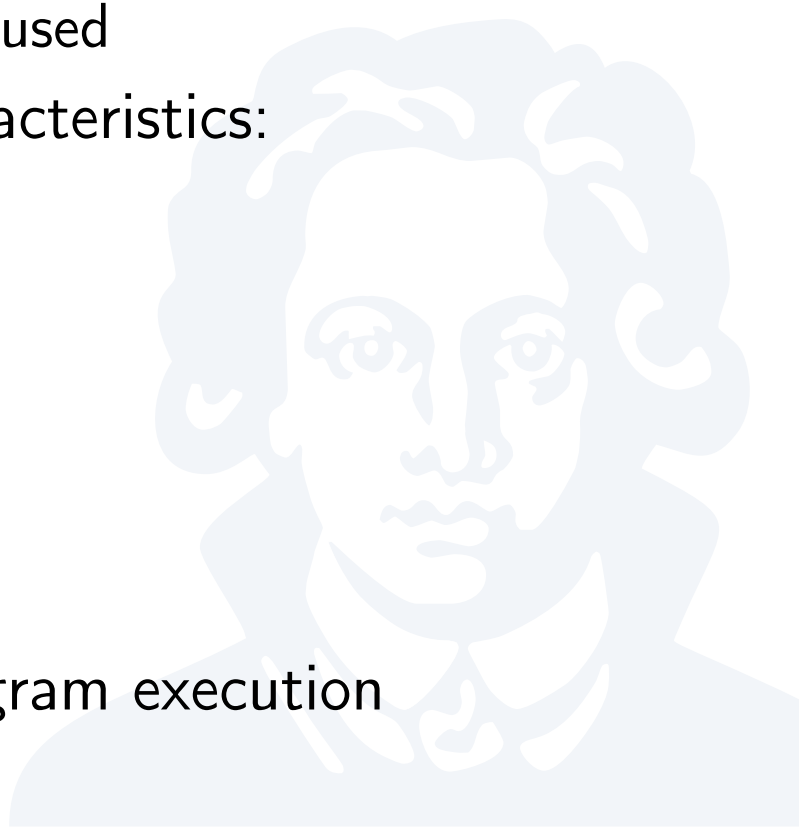
- Execute unit tests
- Detect: do we actually use instances of C in HashMap/HashSet?
- Detect: do we observe (for any a, b of C) that

$$\texttt{a.equals(b)} \text{ and } \texttt{a.hashCode()} \neq \texttt{b.hashCode()}$$

Example of *combined static/dynamic analysis*

Summary: Dynamic Analysis

- Dynamic analysis:
  - Is incomplete
  - Is unsound
  - Can reveal how code is *actually* used
- Many ways to measure event characteristics:
  - Total counts
  - Various sampling techniques
  - Emulation
  - Performance counters
  - ...
- Data is often imprecise
- Can offer unique insights into program execution



Thanks for listening
Good luck with your projects!