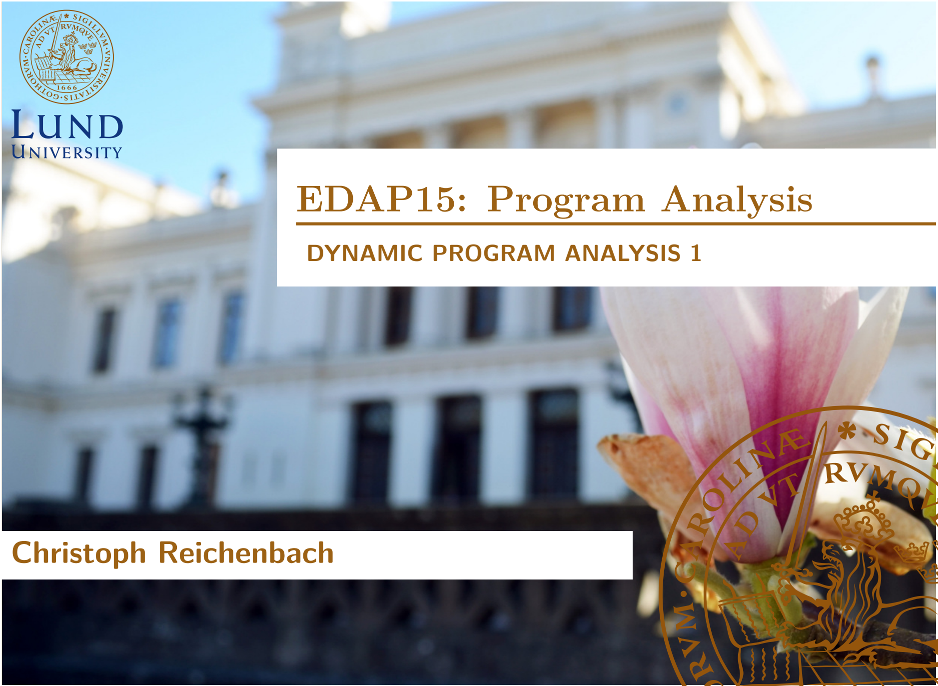


LUND UNIVERSITY

EDAP15: Program Analysis

DYNAMIC PROGRAM ANALYSIS 1

Christoph Reichenbach

Welcome back!

- Homework assignments relaxation:
  - Due to TA sickness, some students did not get timely feedback
  - Will announce options today or tomorrow
- Questions?

Static Analysis: Limitations

- Static program analysis faces significant challenges:
  - **Decidability** requires lack of precision or soundness for most of the interesting analyses
  - **Reflection** allows calling methods / creating objects given by arbitrary string
  - **Dynamic module loading** allows running code that the analysis couldn't inspect ahead of time
  - **Native code** allows running code written in a different language
  - **Dynamic code generation** and `eval` allow building arbitrary programs and executing them
  - No universal solution
  - Can try to 'outlaw' or restrict problematic features, depending on goal of analysis
  - Can combine with dynamic analyses

More Difficulties for Static Analysis

- Does a certain piece of code actually get executed?
- 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?
  - Harddisks?
  - Networking devices?
  - *Caches* that speed up memory access?
  - *Branch predictors* that speed up conditional jumps?
  - The *ALU(s)* that perform arithmetic in the CPU?
  - The *TLB* that helps look up memory?

  ...

Impossible to predict for all practical situations

Static vs. Dynamic Program Analyses

	Static Analysis	Dynamic Analysis
Examines	Program structure	Program execution
Input	Independent	Dependent
Hardware/OS	Independent	Dependent (for some properties)
Perspective	Sees anything that *could* happen	Sees that which *does* happen
False Negatives	*Avoidable*	Need all possible inputs
False Positives	Unavoidable	*Avoidable*

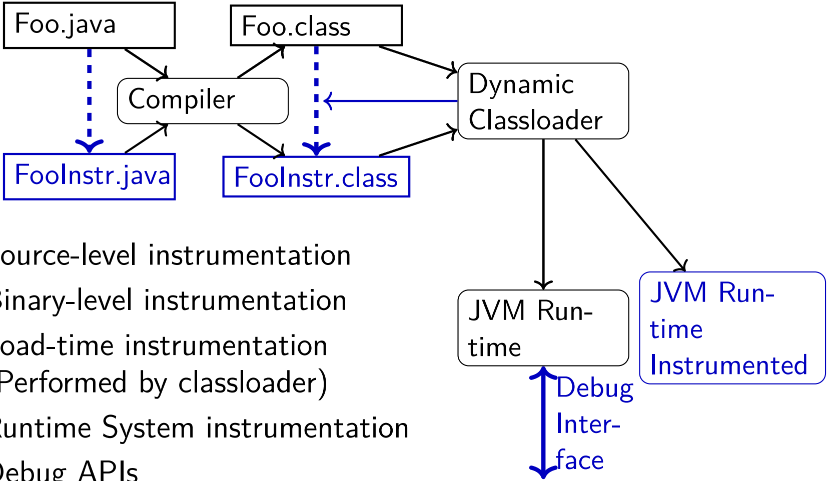
Summary

- Static analysis has key limitations:
  - *Information missing from code* (cf. *Soundiness*)
  - *Dependency on hardware details* (e.g. *Execution Time*))
- This limits:
  - Optimisation: *which optimisations are worthwhile?*
  - Bug search: *which potential bugs are 'real'?*
- Can use *dynamic analysis* to examine run-time behaviour

Gathering Dynamic Data

- **Instrumentation**
- Performance Counters
- Emulation

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
  - − Must handle syntactic issues, name capture, ...
  - − Only applicable if we have all source code
- **Binary-level instrumentation**:
  - + Flexible
  - − Must handle binary encoding issues
  - − Only applicable if we know what binary code is used
- **Load-time instrumentation**:
  - + Flexible
  - + Can handle even unknown code
  - − Requires run-time support, may clash with custom loaders
- **Runtime system instrumentation**:
  - + Flexible
  - + Can see everything (gc, JIT, ...)
  - − Labour-intensive and error-prone
  - − Becomes obsolete quickly as runtime evolves
- **Debug APIs**:
  - + Typically easy to use and efficient
  - − 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)
- *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**

Unit Tests

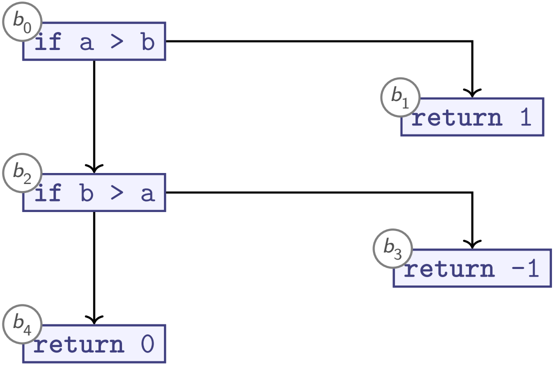
Teal

```
fun cmp(a, b) = {
  if a > b {
    return 1;
  }
  if a < b {
    return -1;
  }
  return 0;
}

fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

Unit tests are a simple form of dynamic program analysis

Unit Test Quality

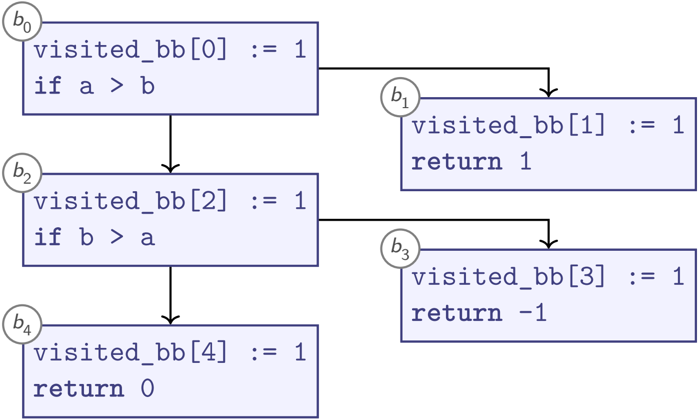


Teal

```
fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

Have I tested all behaviours?

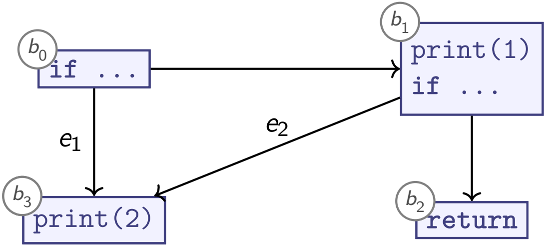
Test Coverage



- Test coverage = fraction of `visited_bb` elements updated

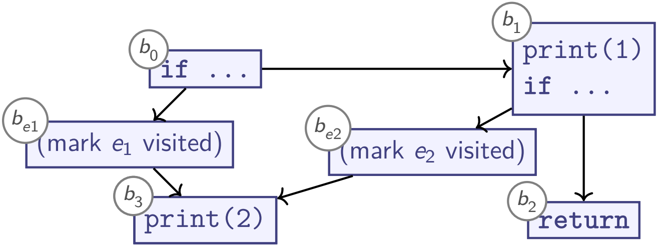
Test Coverage Properties

- **Statement Coverage**: % of executed Basic Blocks
  - Mark basic blocks as visited while testing
- **Edge Coverage**: % of taken CFG edges
  - Challenge: distinguish *edges* e_1 from e_2?



Test Coverage Properties

- **Statement Coverage**: % of executed Basic Blocks
  - Mark basic blocks as visited while testing
- **Edge Coverage**: % of taken CFG edges
  - Challenge: distinguish *edges* e_1 from e_2?



- **Path Coverage**: % of CFG paths
  - Must limit iterations
  - Must restart tracking block coverage on every method entry

Summary

- **Unit Tests** are a simple form of dynamic program analysis
  - Minimal tooling needed
  - Custom checks
  - Limited to what underlying language can express directly
- **Test Coverage** tells us how much of our code gets analysed by at least one unit test
- Implement by setting markers on relevant basic blocks
- Different criteria, such as:
  - **Statement Coverage**
  - **Edge Coverage**: may require helper BBs
  - **Path Coverage**: paths through CFG (usually excluding loops)
- Tools for Java: JCov, JaCoCo

General Data Collection

- *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

- *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

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 current time again, update aggregate
- Reading timer takes: ~ 80 cycles
- Short `f` calls may be much faster than 160 cycles
- Also: measurement needs CPU registers
  $\Rightarrow$ may require registers
  $\Rightarrow$ may slow down code further

Measurements perturb our results, slow down execution

Sampling

Alternative to full counts: *Sampling*

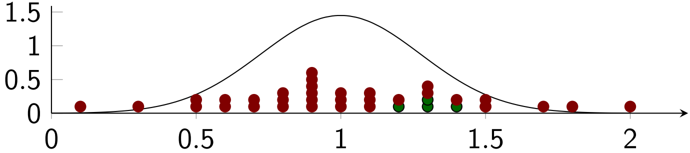
- Periodically interrupt program and measure
- Problem: how to pick the right period?

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

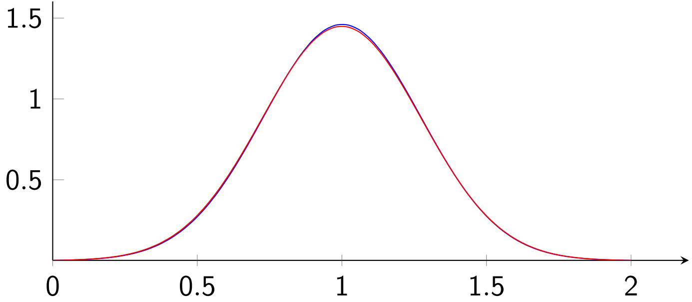
Summary

- 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

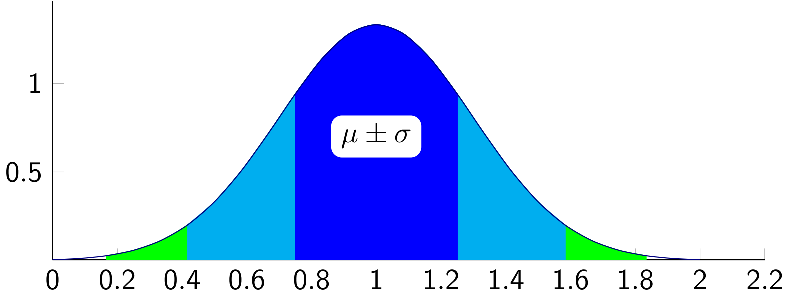
Presenting Measurements

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

Assuming normal distribution:



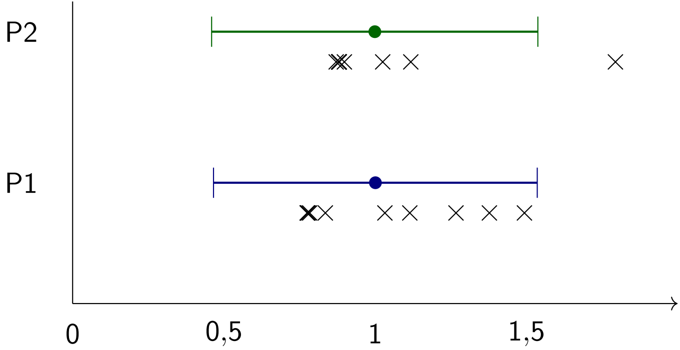
Standard Deviation, Assuming Normal Distribution



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%

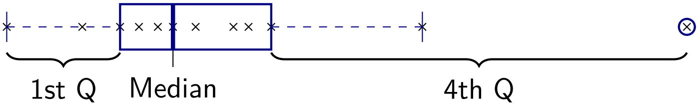
How Well Does Normal Distribution Fit?

Representation with error bars (95% confidence interval):



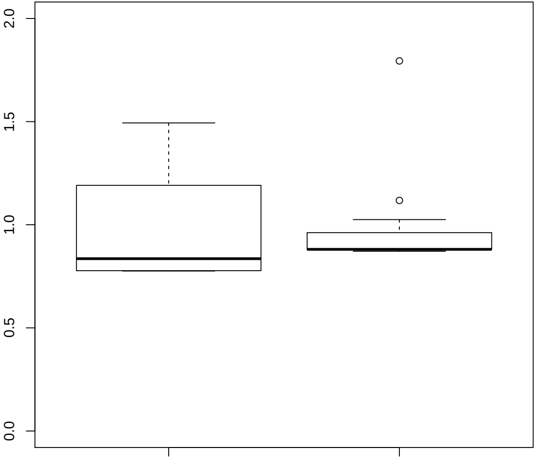
Mean + Std.Dev. are misleading if measurements don't observe normal distribution!

Box Plots

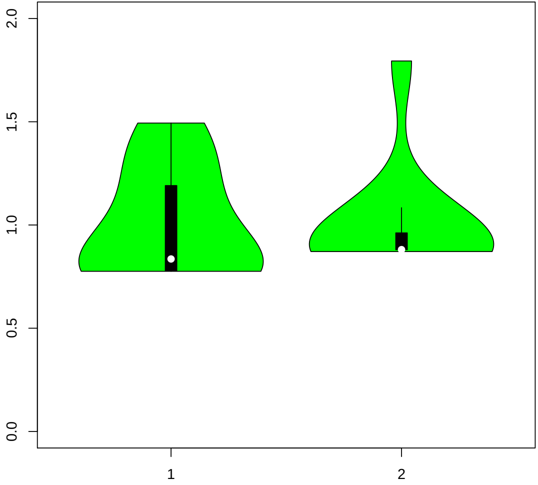


- Split data into 4 *Quartiles*:
  - Upper Quartile (1st Q): Largest 25% of measurements
  - Lower Quartile (4th Q): Smallest 25% of measurements
  - Median: measured value, middle of sorted list of measurements
- Box: Between 1st/4th quartile boundaries
  Box width = inter-quartile range (*IQR*)
- 1st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

Automatic Performance Measurement

- Profiler:
  - Interrupts program during execution
  - Examines call stack
- Simulator:
  - Simulates CPU/Memory in software
  - Tries to replicate inner workings of machine
  - Often also an *Emulator* (= replicate observable functionality)
- Operating System:
  - Counts important system events (network accesses etc.)
- CPU:
  - Hardware performance counters count interesting events

Profiler

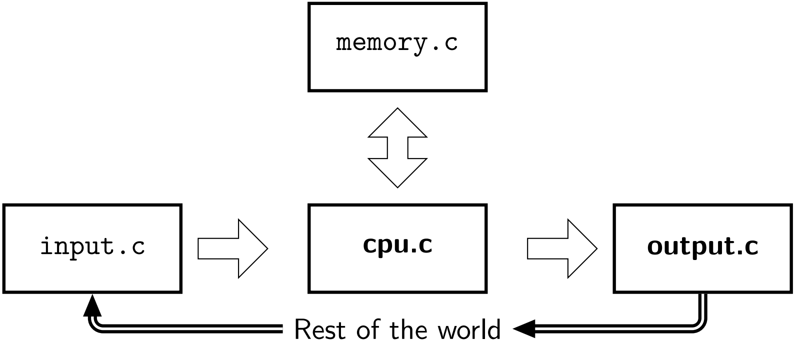
- Measures: which functions are we spending our time in?
- Approach:
  - Build stack maps
  - Execute program, interrupt regularly
  - During interrupt:
    - Examine stack
- Infer functions from stack contents

Execution Stack

`return` (alt-1)
`$fp` (alt-1)
...
...
`return` (alt-2)
`$fp` (alt-2)
...

Can be inaccurate: misses short function calls

Simulator



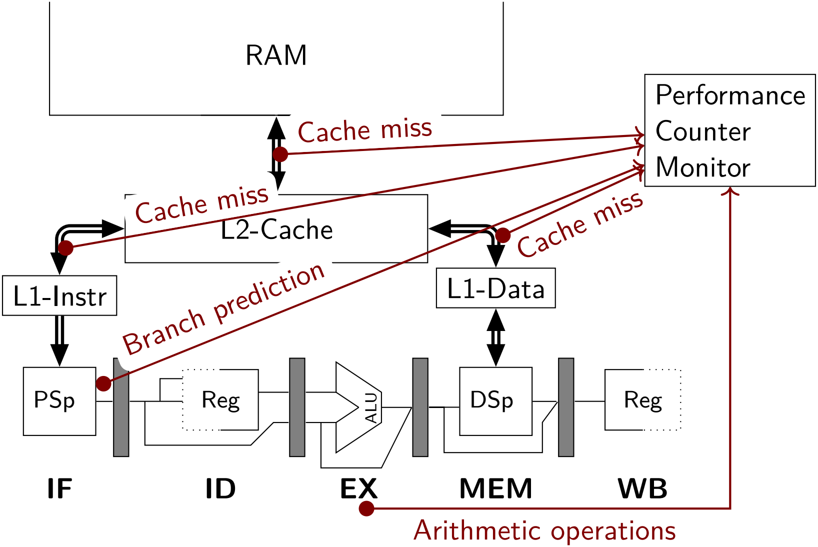
- Software simulates hardware components
- Can count events of interest (memory accesses etc.)

Modern CPUs are very complex: Simulators tend to be inaccurate

Software Performance Counters

- Complex software may use high-level properties such as:
  - How much time do we spend waiting for the harddisk?
  - How often was our program suspended by the operating system in order to let another program run?
  - How much data did we receive through the network?
- Operating systems collect many of these statistics

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

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.)

...

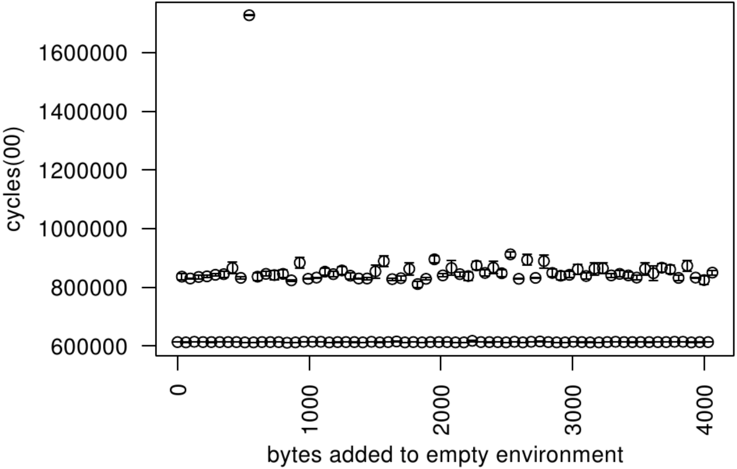
Is that all?

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



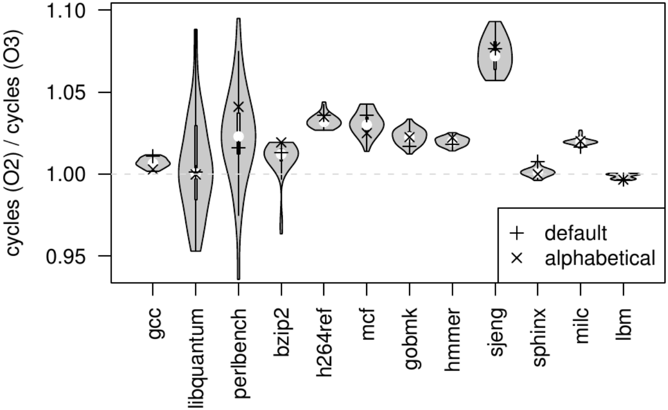
Mytkowicz, Diwan, Hauswirth, Sweeney: "Producing wrong data without doing anything obviously wrong", in ASPLOS 2009

Linking Order

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

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

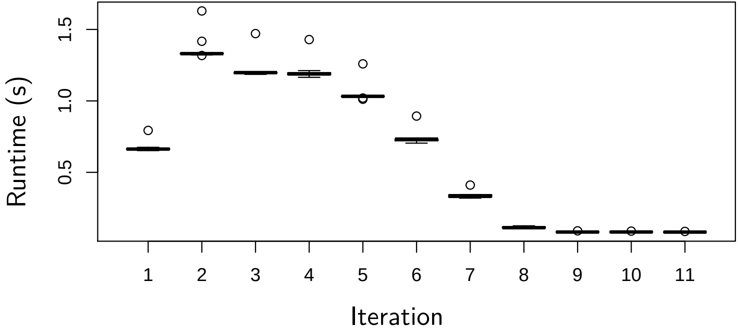
```
gcc b.o a.o -o program
```
(Variant 2)



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

Adaptive Systems

- Measurement: 11 runs



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*
- What should we measure?
  - Latency: measure first run
    Reset system before every run
  - Throughput: later runs
    Discard initial *n* measurements

Ignored Parameters

- Performance affected by subtle effects
- System 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

    ...
- 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*

Outlook

- Guest Lecture on Wednesday:
  - **Noric Couderc** (LTH): Bayesian Methods for Dynamic Program Analysis
- Guest Lecture next Monday (first half):
  - **Patrik Åberg**, **Magnus Templing** (Ericsson): DMCE: Did My Code Execute?

`http://cs.lth.se/EDAP15`