Reliable Benchmarking: Requirements and Solutions

Dirk Beyer
Joint Work with Stefan Löwe and Philipp Wendler



LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

Evaluation of Research Result

- Result “Theorem”
  Evaluation “Proof”
- Result “Algorithm”
  Evaluation “Algorithm Analysis, properties, Big-O”
- Result “Heuristics for Complex Problems”
  Evaluation “Performance Experiments”

Comparative Evaluation

- Old: Done by competitors
- New: Done by independent competitions

Notions from Experimental Research

Experimental science needs:

Repeatability
: Same team, same experimental setup

Replicability
: Different team, same experimental setup

Reproducibility
: Different team, different experimental setup

Source:
https://www.acm.org/publications/policies/artifact-review-badging

Notions from Experimental Research

Example: You implemented new algorithm in CPAchecker and compared it against Ultimate.

Repeatability

You execute same version of CPAchecker again. Are the numbers the same?

Replicability

Somebody else takes same version of CPAchecker and benchmark set and executes it.

Reproducibility

Somebody implements both algorithms in a different tool (e.g., Ultimate) and compares them.

Notions from Experimental Research

Repeatability
: Can you produce the same results
for the camera-ready version again?

Replicability
: Can others take your tool etc.
and perform the experiment?
(main goal of providing artifacts)

Reproducibility
: Can others come to the same conclusion
in a different experiment?

Background: Wording

experiments	can be replicable
experiments	can be repeatable (weaker than replicable)
effects	can be reproducible
conclusions	can be reproducible
performance results	can be replicable (but better avoid this)

Background: Wording

experiments can be replicable

experiments can be repeatable (weaker than replicable)

effects can be reproducible

conclusions can be reproducible

performance results can be replicable (but better avoid this)

measurements can be accurate and precise

benchmarking can be reliable

runs are executed

Background: Wording

experiments can be replicable

experiments can be repeatable (weaker than replicable)

effects can be reproducible

conclusions can be reproducible

performance results can be replicable (but better avoid this)

measurements can be accurate and precise

benchmarking can be reliable

runs are executed

We avoid

- benchmark
- to run

Background: Requirements

Repeatability

- everything documented
  (machine, version of tool and OS, parameters)
- deterministic tool
- **reliable benchmarking**

Replicability

- everything above
- availability of tool, benchmark set,
  configuration, environment
  (published and archived, appropriate license)

Reproducibility

(not discussed here)

Benchmarking is Important

- Evaluation of new approaches
- Evaluation of tools
- Competitions
- Tool development (testing, optimizations)

Reliable, replicable, and accurate results needed!

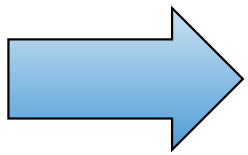
Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Not relevant for
most verification tools



Easy?

Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Not relevant for most verification tools

- Different hardware architectures
- Heterogeneity of tools
- Parallel benchmarks

Relevant!

Goals

- Replicability
  - Avoid non-deterministic effects and interferences
  - Provide defined set of resources
- Accurate results
- For verification tools (and similar)
- On Linux

Checklist

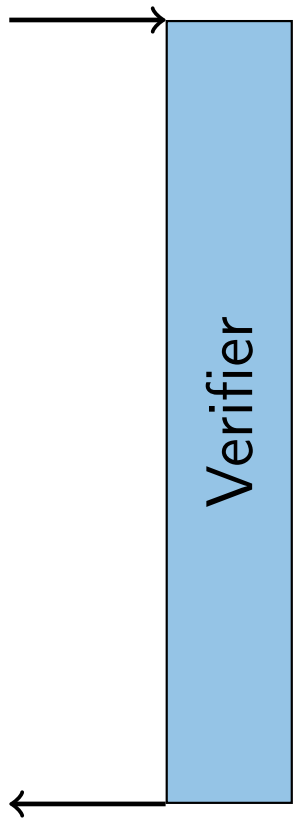
1. Measure and Limit Resources Accurately
   - Time
   - Memory
2. Terminate Processes Reliably
3. Assign Cores Deliberately
4. Respect Non-Uniform Memory Access
5. Avoid Swapping
6. Isolate Individual Runs
   - Communication
   - File system

Measure and Limit Resources Accurately

- Wall time and CPU time
- Define memory consumption
  - Size of address space? Too large
  - Size of heap? Too low
  - Size of resident set (RSS)?
- Measure peak consumption
- Always define memory limit for replicability
- Include sub-processes

Measuring CPU time with "`time`"

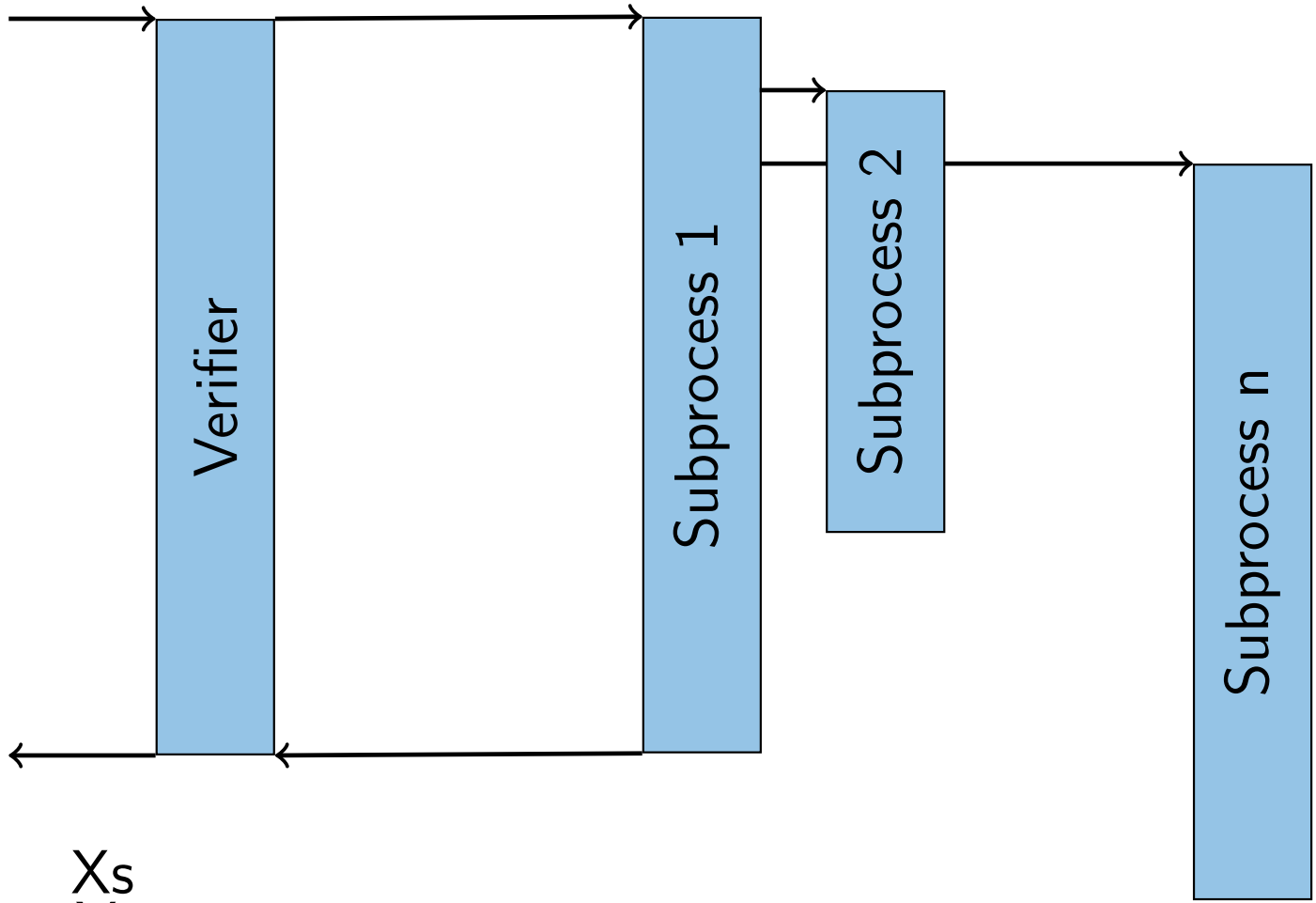
```
~$ time verifier
```



real	Xs
user	Ys
sys	Zs

Measuring CPU time with "`time`"

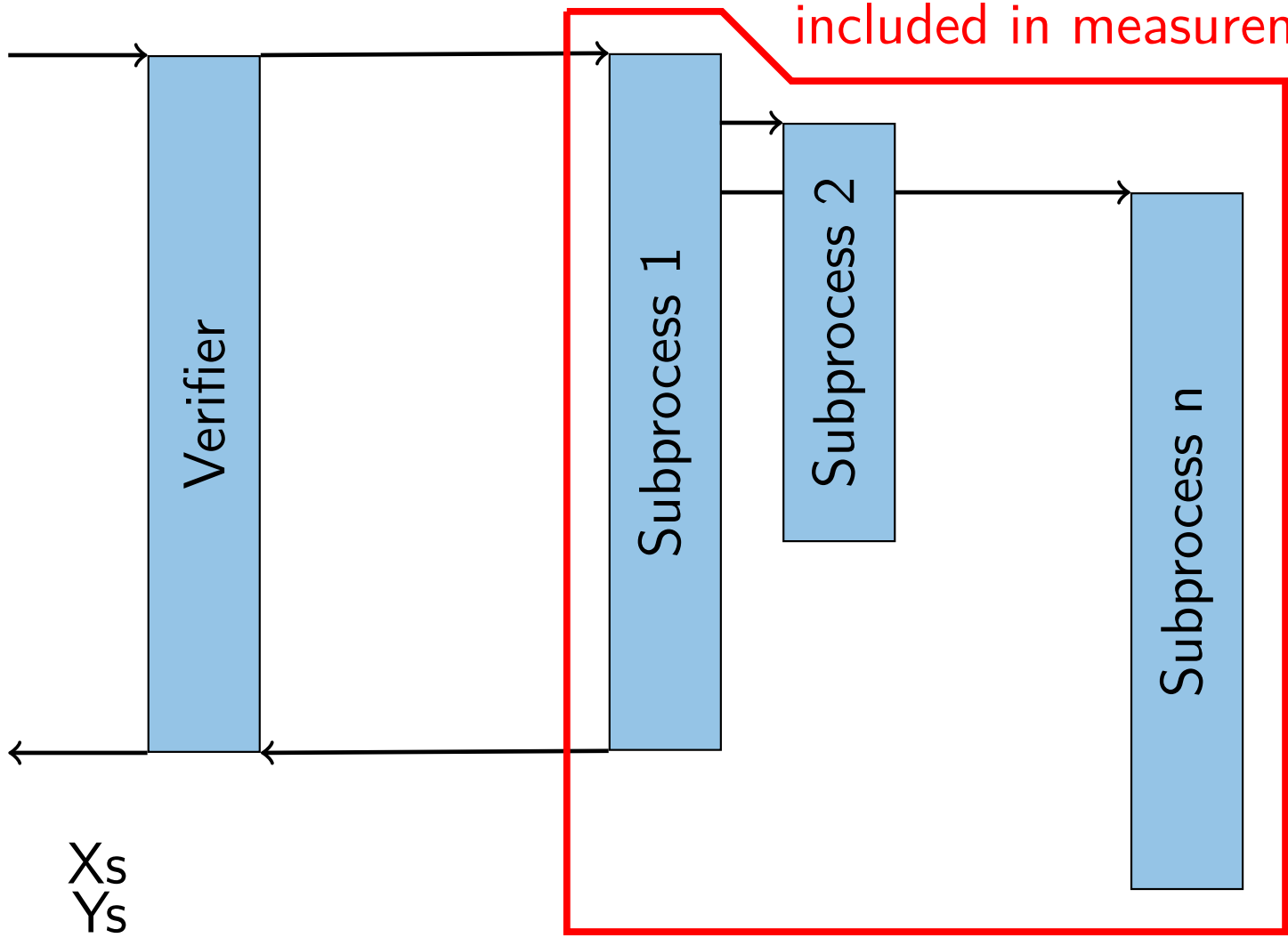
```
~$ time verifier
```



real Xs
user Ys
sys Zs

Measuring CPU time with "time"

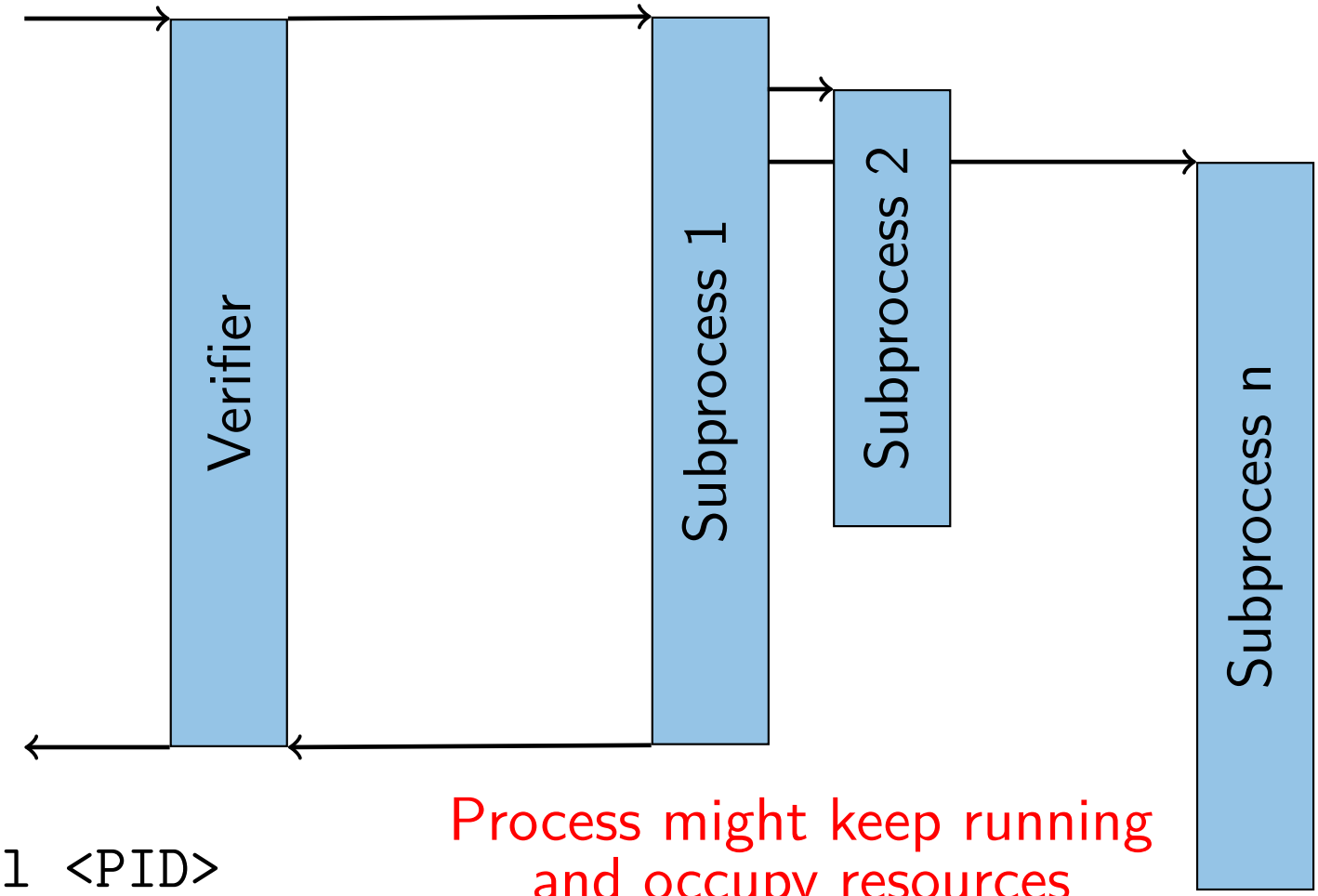
```
~$ time verifier
```



real Xs
user Ys
sys Zs

Terminate Processes Reliably

```
~$ verifier
```




```
~$ kill <PID>
```

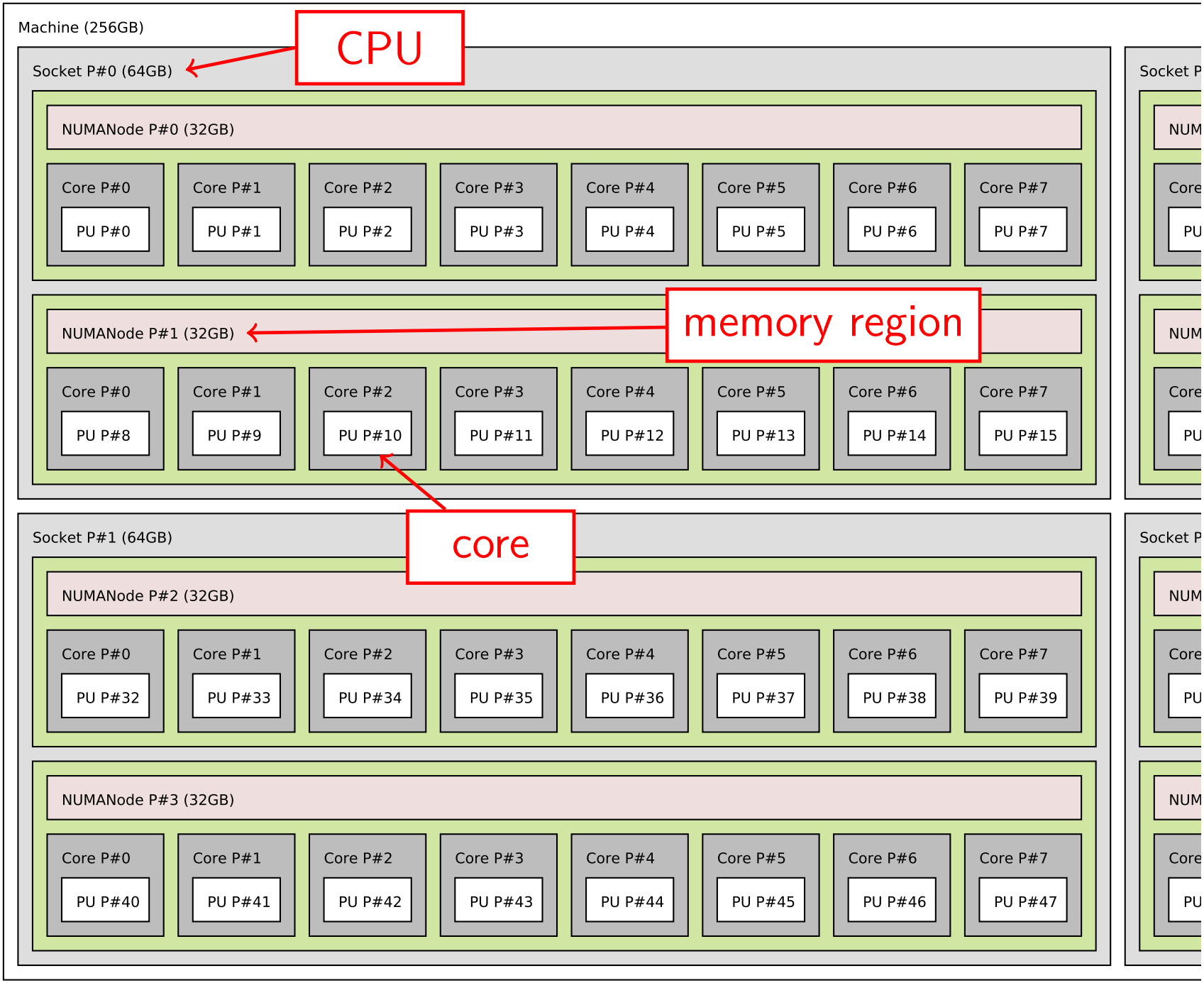
Process might keep running
and occupy resources

Assign Cores Deliberately

- Hyper Threading:
  Multiple threads sharing execution units
- Shared caches

Respect Non-Uniform Memory Access (NUMA)

- Memory regions have different performance depending on current CPU core
- Hierarchical NUMA makes things worse


Machine (256GB)
CPU
Socket P#0 (64GB)
NUMANode P#0 (32GB)
Core P#0
PU P#0
Core P#1
PU P#1
Core P#2
PU P#2
Core P#3
PU P#3
Core P#4
PU P#4
Core P#5
PU P#5
Core P#6
PU P#6
Core P#7
PU P#7
NUMANode P#1 (32GB)
memory region
Core P#0
PU P#8
Core P#1
PU P#9
Core P#2
PU P#10
Core P#3
PU P#11
Core P#4
PU P#12
Core P#5
PU P#13
Core P#6
PU P#14
Core P#7
PU P#15
core
Socket P#1 (64GB)
NUMANode P#2 (32GB)
Core P#0
PU P#32
Core P#1
PU P#33
Core P#2
PU P#34
Core P#3
PU P#35
Core P#4
PU P#36
Core P#5
PU P#37
Core P#6
PU P#38
Core P#7
PU P#39
NUMANode P#3 (32GB)
Core P#0
PU P#40
Core P#1
PU P#41
Core P#2
PU P#42
Core P#3
PU P#43
Core P#4
PU P#44
Core P#5
PU P#45
Core P#6
PU P#46
Core P#7
PU P#47

Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # ... (tool started here)

  ```
  killall z3 2> /dev/null
  killall minisat 2> /dev/null
  killall yices 2> /dev/null
  ```

- Thanks for thinking of cleanup

Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # ... (tool started here)

```
killall z3 2> /dev/null
killall minisat 2> /dev/null
killall yices 2> /dev/null
```

- Thanks for thinking of cleanup
- But what if there are parallel runs?

Isolate Individual Runs

- Temp files with constant names like `/tmp/mytool.tmp` collide
- State stored in places like `~/.mytool` hinders reproducibility
  - Sometimes even auto-generated
- Restrict changes to file system as far as possible

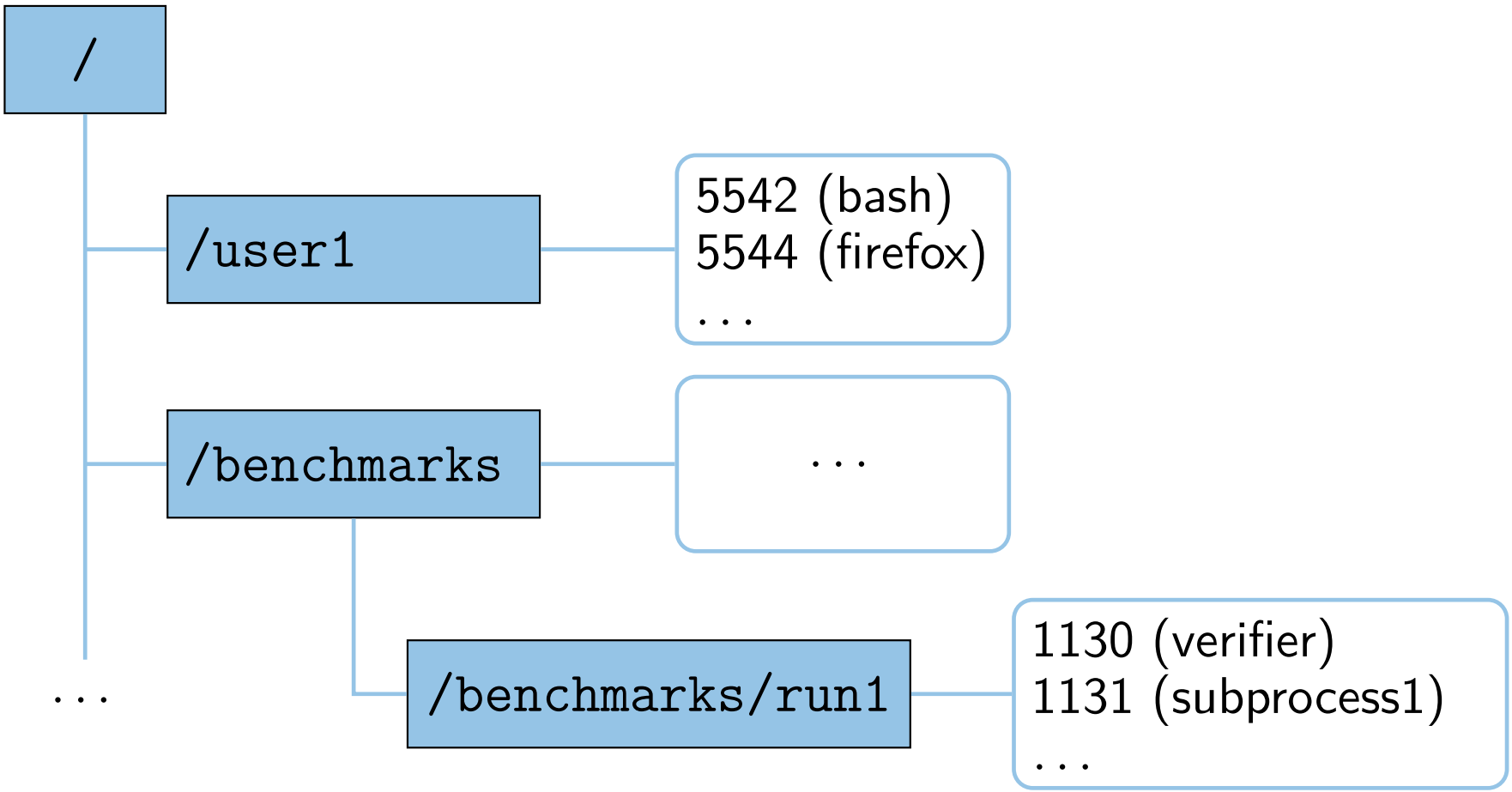
Cgroups

- Linux kernel “control groups”
- Reliable tracking of spawned processes
- Resource limits and measurements per cgroup
  - CPU time
  - Memory
  - I/O etc.

Only solution on Linux
for race-free handling of multiple processes!

Cgroups

- Hierarchical tree of sets of processes



Namespaces

- Light-weight virtualization
- Only one kernel running, no additional layers
- Change how processes see the system
- Identifiers like PIDs, paths, etc. can have different meanings in each namespace
  - PID 42 can be a different process in each namespace
  - Directory / can be a different directory in each namespace
  - ...
- Can be used to build application containers without possibility to escape
- Usable without root access

Benchmarking Containers

- Encapsulate groups of processes
- Limited resources (memory, cores)
- Total resource consumption measurable
- All other processes hidden
  and no communication with them
- Disabled network access
- Adjusted file-system layout
  - Private `/tmp`
  - Writes redirected to
    temporary RAM disk

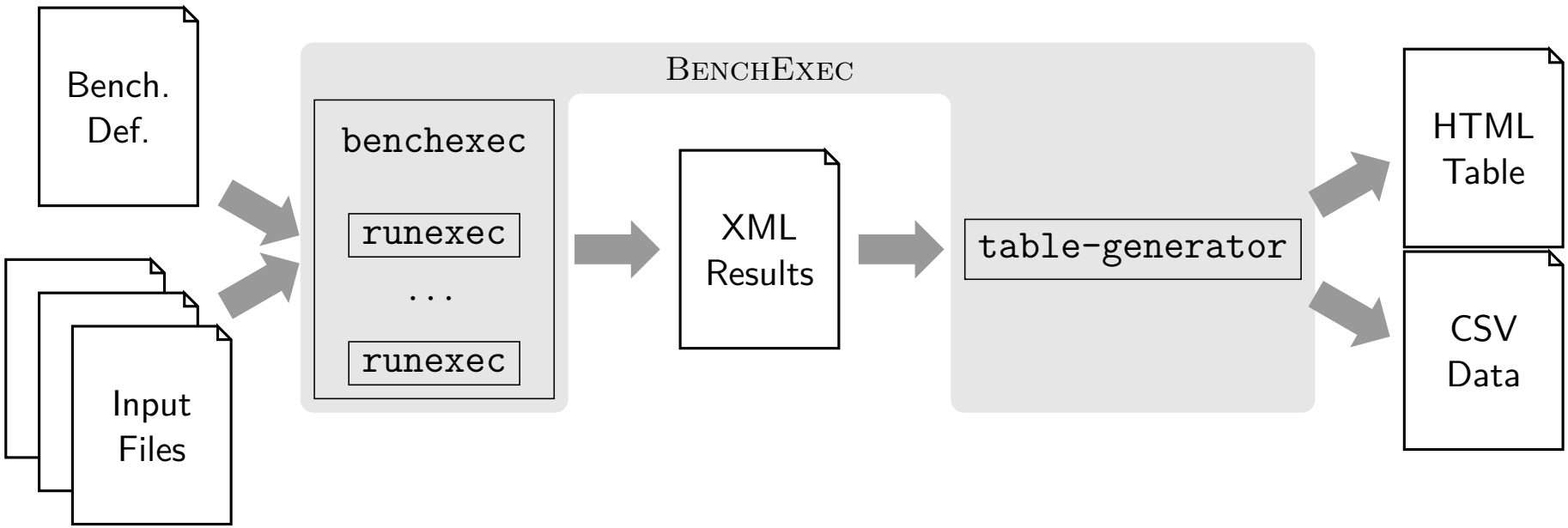
BenchExec

- A Framework for Reliable Benchmarking and Resource Measurement
- Provides benchmarking containers based on cgroups and namespaces
- Allocates hardware resources appropriately
- Low system requirements (modern Linux kernel and cgroups access)

BenchExec

- Open source: Apache 2.0 License
- Written in Python 3
- https://github.com/sosy-lab/benchexec
- Used in International Competition on Software Verification (SV-COMP) and by StarExec
- Originally developed for software-verification, but applicable to arbitrary tools

BenchExec Architecture



`runexec`

Benchmarks a single run of a tool (in container)

`benchexec`

Benchmarks multiple runs

`table-generator`

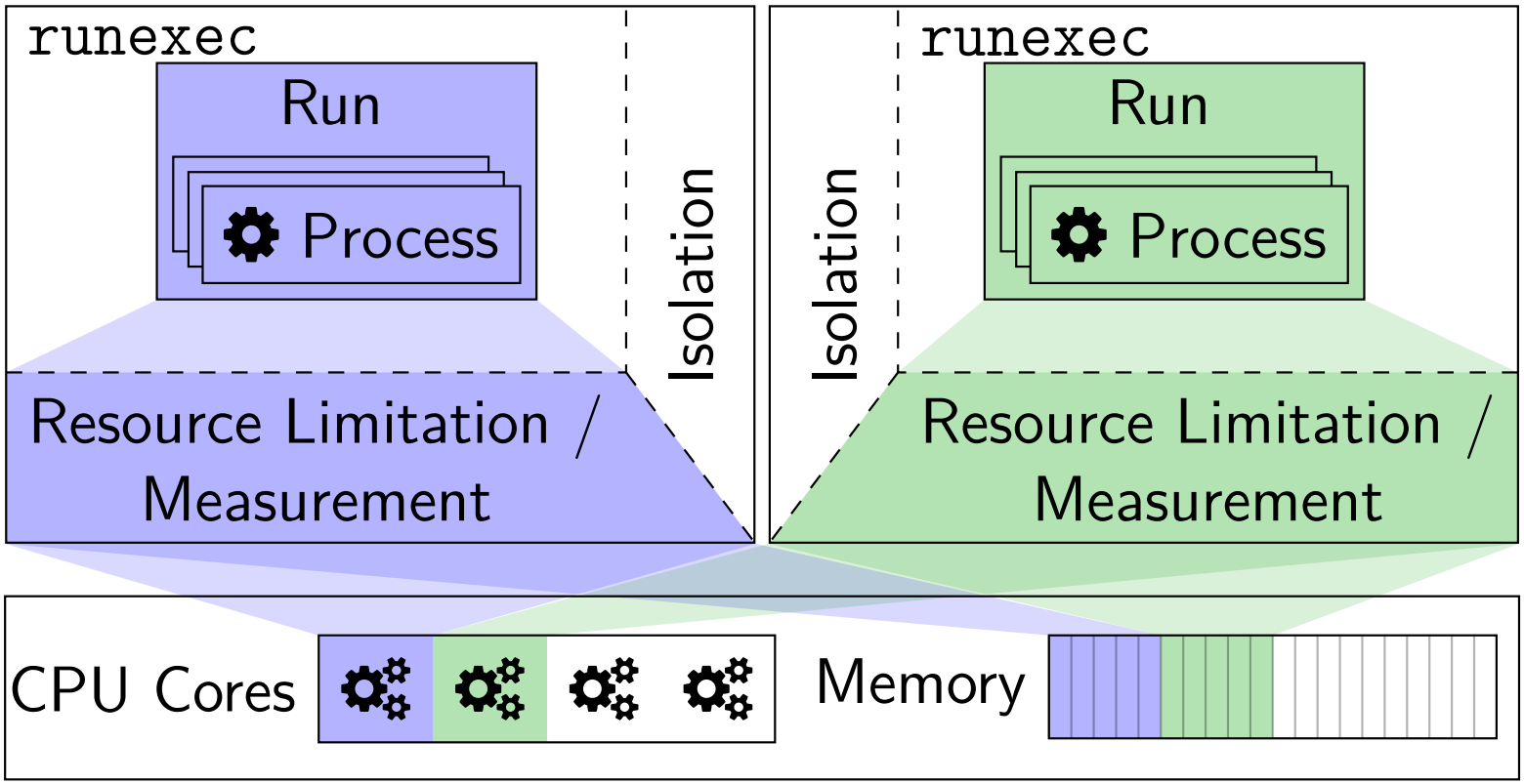
Generates CSV and interactive HTML tables

BenchExec: `runexec`

- Benchmarks a single run of a tool
- Measures and limits resources using cgroups
- Runnable as stand-alone tool and as Python module
- Easy integration into other benchmarking frameworks and infrastructure
- Example:

```
runexec --timelimit 100 --memlimit 16000000000
        --cores 0-7,16-23 --memoryNodes 0
        --<TOOL_CMD>
```

BenchExec: `runexec`



BenchExec: `benchexec`

- Benchmarks multiple runs
  (e.g., a set of configurations against a set of files)
- Allocates hardware resources
- Can check whether tool result is as expected
  for given input file and property

BenchExec: `table-generator`

- Aggregates results
- Extracts statistic values from tool output
- Generates CSV and interactive HTML tables (with plots)
- Computes result differences and regression counts

BenchExec Configuration

- Tool command line
- Expected result
- Resource limits
  - CPU time, wall time
  - Memory
- Container setup
  - Network access
  - File-system layout
- Where to put result files

Conclusion

Be careful when benchmarking!

Don't use time, ulimit etc.
Always use cgroups and namespaces!

BenchExec
https://github.com/sosy-lab/benchexec

There's more

Dirk Beyer, *Stefan Löwe*, and *Philipp Wendler*.
Reliable Benchmarking:
Requirements and Solutions. [1]
STTT 2019 (preprint available here)

- More details
- Study of hardware influence on benchmarking results
- Suggestions how to present results
  (result aggregation, rounding, plots, etc.)

References I

Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y