Int J Softw Tools Technol Transfer (2019) 21:1–29
https://doi.org/10.1007/s10009-017-0469-y

REGULAR PAPER

Reliable benchmarking: requirements and solutions

Dirk Beyer[1] · **Stefan Löwe**[2] · **Philipp Wendler**[1]

Published online: 3 November 2017
© Springer-Verlag GmbH Germany 2017

Abstract Benchmarking is a widely used method in experimental computer science, in particular, for the comparative evaluation of tools and algorithms. As a consequence, a number of questions need to be answered in order to ensure proper benchmarking, resource measurement, and presentation of results, all of which is essential for researchers, tool developers, and users, as well as for tool competitions. We identify a set of requirements that are indispensable for reliable benchmarking and resource measurement of time and memory usage of automatic solvers, verifiers, and similar tools, and discuss limitations of existing methods and benchmarking tools. Fulfilling these requirements in a benchmarking framework can (on Linux systems) currently only be done by using the cgroup and namespace features of the kernel. We developed BenchExec, a ready-to-use, tool-independent, and open-source implementation of a benchmarking framework that fulfills all presented requirements, making reliable benchmarking and resource measurement easy. Our framework is able to work with a wide range of different tools, has proven its reliability and usefulness in the International Competition on Software Verification, and is used by several research groups worldwide to ensure reliable benchmarking. Finally, we present guidelines on how to present measurement results in a scientifically valid and comprehensible way.

Keywords Benchmarking · Resource measurement · Process control · Process isolation · Container · Competition

A preliminary version was published in Proc. SPIN 2015 [10].

✉ Philipp Wendler

1 LMU Munich, Munich, Germany

2 One Logic, Passau, Germany

1 Introduction

Performance evaluation is an effective and inexpensive method for assessing research results [31], and in some communities, like high-performance computing,[1] transactional processing in databases,[2] natural-language requirements processing,[3] and others, performance benchmarking is standardized. Tools for automatic verification, such as solvers and verifiers, are also evaluated using performance benchmarking, i.e., measuring execution time, memory consumption, and other performance characteristics of the tool for a large set of input files. Benchmarking is used for comparing different tools of the same domain, evaluating and comparing different features or configurations of the same tool, or finding out how a single tool performs on different inputs or during regression testing. Also competitions, like the SAT Competition (SAT-COMP) [1], the International Satisfiability Modulo Theories Competition (SMT-COMP) [13], and the International Competition on Software Verification (SV-COMP) [7], require exact measuring of resource consumption for hundreds or thousands of runs for each participating tool.

Besides *measuring*, also the ability to *limit* resource usage (e.g., memory consumption) of a tool during benchmarking is a hard requirement for replicable experiments. Competitions need to accurately enforce the agreed resource limits in order to guarantee fairness. For example, SV-COMP limits all tools to 15 min of CPU time and 15 GB of RAM [7]. Results from the tools are only counted if none of these limits are exceeded.

To recover from its replication crisis [14,23,26], experimental computer science needs a stronger focus on

1 https://www.spec.org

2 http://www.tpc.org

3 http://nlrp.ipd.kit.edu

replicability [11,20,33]. While replicability requires several properties to be satisfied (e.g., documentation of experimental setup, availability of data, repeatability of experiments), this article focuses on the technical aspects of benchmarking that can render the results invalid. To be able to perform replicable performance experiments, a benchmarking infrastructure should guarantee that the results are obtained by reliable and valid measurements.

An experiment is *replicable* if it is guaranteed that a different research team is able to obtain the same results later again by rerunning the benchmarks on a machine with the same hardware and the same software versions.[4] Replicable experiments require reliable *measurement*. We call a measurement *reliable*, if the measurement method ensures high *accuracy* (only small systematic and random measurement error, i.e., no bias or "volatile" effects, resp.) and sufficient *precision* [19] (cf. also ISO 3534-2:2006). While measuring execution time may appear trivial, a closer look reveals that quite the contrary is the case. In many circumstances, measuring the wall time, i.e., the elapsed time between start and end of a tool execution, is insufficient because this does not allow a meaningful comparison of the resource usage of multithreaded tools and may be inadvertently influenced by input/output operations (I/O). Measuring the CPU time is more meaningful but also more difficult, especially if child processes are involved. Furthermore, characteristics of the machine architecture such as hyperthreading or nonuniform memory access can *nondeterministically* affect results and need to be considered carefully in order to obtain accurate results. Obtaining reliable measurement values on memory consumption is even harder, because the memory that is used by a process may increase or decrease at any point in time. Similarly, the *limits* on memory consumption must not be exceeded *at any point in time* during the execution of the tool. Child processes again add further complications. For replicable experiments, it is also important that the tool executions are properly *isolated*. Executing the benchmarked tool should leave the system in an unchanged state, and parallel tool executions must not affect each other due to, for example, mutual interference of processes from different tool executions or contention with regard to shared hardware resources. Another important aspect is the potentially huge heterogeneity between different tools in a comparison: tools are written in different programming languages, require different libraries, may spawn child processes, write to storage media, or perform other I/O. All of this has to be considered in the design of a benchmarking environment, ideally in a way that does not exclude any tools from being benchmarked.

Unfortunately, existing benchmarking tools do not always ensure that these issues are handled and that the results are reliable. In order to pave the way for reliable benchmarking in practice, a benchmarking infrastructure is necessary that allows reliable measurement and limitation of resources and makes it as easy as possible to obtain and present scientifically valid experimental data.

The presentation of benchmarking results can also introduce errors that reduce the precision, for example due to incorrect rounding. Furthermore, presenting a large number of results, such as for experiments with several tool configurations on many input files, in a meaningful and understandable way can be a challenge.

1.1 Overview

We present the following solutions and insights toward reliable benchmarking for all scenarios that are described above:

- We define a set of necessary requirements that need to be fulfilled for reliable benchmarking (Sect. 2).
- We show that some existing methods for resource measurements and limitations do not fulfill these requirements and lead to invalid experimental results in practice (Sect. 3).
- We investigate the impact of benchmarking multiple tool executions in parallel and report experimental results on measurement errors, depending on certain hardware characteristics (Sect. 4).
- We describe how to implement a benchmarking environment on a Linux system which fulfills all mentioned requirements (Sect. 5).
- We introduce the open-source implementation BenchExec, a set of ready-to-use tools that fulfill the requirements for reliable benchmarking. These tools were already used successfully in practice by competitions and by research groups (Sect. 6).
- We identify requirements for valid presentation of experimental results and give an overview of methods for presenting benchmark results using tables and expressive figures (Sect. 7).

Compared to our previous publication on this topic [10], we describe how to create benchmarking containers with Linux namespaces (Sect. 5.2) to ensure the desired isolation of benchmark runs (Sect. 2.6); we show results for the performance influence of several hardware characteristics and present new insights on the validity of executing benchmarks in parallel (Sect. 4); and we add the section on presentation of benchmark results (Sect. 7).

1.2 Restrictions

In order to guarantee reliable benchmarking, we need to introduce a few restrictions. There are important classes of

[4] cf. ACM's guideline:
https://www.acm.org/publications/policies/artifact-review-badging

benchmarks in the area of efficient algorithms for which these restrictions are acceptable, for example automatic solvers, verifiers, and similar tools. We only consider the benchmarking of tools that adhere to the following restrictions: the tool (1) is CPU-bound, i.e., if compared to CPU usage, input and output operations from and to storage media are negligible, and input and output bandwidth does not need to be limited nor measured (this assumes the tool does not make heavy use of temporary files); (2) executes computations only on the CPU, i.e., does not make use of separate coprocessors such as GPUs, (3) does not require external network communication during the execution; (4) does not spread across several machines during execution, but is limited to a single machine; and (5) does not require user interaction.

These restrictions are acceptable for a large class of benchmarks. Reading from storage media (1), apart from the input file, is not expected for tools in the target domains. In case a tool produces much output (e.g., by creating large log files), this would primarily have a negative impact on the performance of the tool itself and thus does not need to be restricted by the benchmarking environment. Sometimes, I/O cannot be avoided for communicating between several processes; however, for performance this should be done without any actual storage I/O anyway (e.g., using pipes). Note that RAM disks should not be used as temporary storage on benchmarking systems, because their usage would neither be counted in memory measurements nor be restricted by the memory limit. If a tool executes computations on coprocessors like GPUs (2), this kind of resource consumption would also need to be measured in order to get meaningful results. This is out of our scope, and also more complex than measurements on CPUs, because GPUs and similar coprocessors are architecturally much more diverse than CPUs and no common hardware-independent interface exists for them, yet. Not supporting external network communication (3) is necessary, because allowing such communication would make it possible for a tool to offload work to remote servers [8,32], and this would mean to exclude the offloaded work from benchmarking. Benchmarking a distributed tool (4) is much more complex and out of scope. However, techniques and ideas from this paper as well as our benchmarking framework can be used on each individual host as part of a distributed benchmarking environment. User interaction (5) is generally not supported for benchmarking.

While we consider a proper isolation of the executed tool in order to prevent accidental sabotage of the measurements or other running processes, we do not focus on security concerns, i.e., we assume the executed tool does not maliciously try to interfere with the benchmarking. We also do not consider the task of providing the necessary execution environment, i.e., the user has to ensure that the tool itself and all necessary packages and libraries that are required to execute the tool are available in the correct versions. Furthermore, we assume that enough memory is installed to hold the operating system (OS), the benchmarking environment, and the benchmarked process(es) without the need for swapping, and that no CPU-intensive tasks are running outside the control of the benchmarking environment. All I/O is assumed to be local, because network shares can have unpredictable performance.

These are well-justified requirements, needed for reliable operation of a benchmarking environment, and fulfilled by setups of competitions like SV-COMP.

2 Requirements for reliable benchmarking

There exist three major difficulties that we need to consider for benchmarking. The first problem is that a tool may arbitrarily spawn child processes, and a benchmarking framework needs to control this. Using child processes is common practice. For example, verifiers might start preprocessors, such as `cpp`, or solvers, like an SMT-backend, as child processes. Some tools start several child processes, each with a different analysis or strategy, running in parallel, while some verifiers spawn a separate child process to analyze counterexamples. In general, a significant amount of the resource usage can happen in one or many child processes that run sequentially or in parallel. Even if a tool is assumed not to start child processes, for comparability of the results with other tools it is still favorable to use a generic benchmarking framework that handles child processes correctly.[5]

The second problem occurs if the benchmarking framework assigns specific hardware resources to tool runs, especially if such runs are executed in parallel and the resources need to be divided between them. Machine architectures can be complex and a suboptimal resource allocation can negatively affect the performance and lead to nondeterministic and thus nonreplicable results. Examples for differing machine architectures can be seen in Appendix A and on the supplementary web page[6].

The third problem arises with ensuring the independence of different tool executions. In most cases, benchmarks consist of a large number of tool executions, each of which is considered to be independent from the others. For accurate results, each tool execution should be performed in isolation, as on a dedicated machine without other tool executions, neither in parallel nor in sequential combination, for example to avoid letting the order of tool executions influence the results.

We have identified six requirements (shown in Fig. 1) that address these problems and need to be followed for reliable

[5] Our experience from competition organization shows that developers of complex tools are not always aware of how their system spawns child processes and how to properly terminate them.

[6] https://www.sosy-lab.org/research/benchmarking

1. Measure and Limit Resources Accurately
2. Terminate Processes Reliably
3. Assign Cores Deliberately
4. Respect Nonuniform Memory Access
5. Avoid Swapping
6. Isolate Individual Runs

Fig. 1 Requirements for reliable benchmarking

benchmarking. This list can also serve as a checklist not only for researchers who use benchmarking, but also for assessing the quality of experimental results in research reports. In the following, we explain each requirement in more detail.

2.1 Measure and limit resources accurately

2.1.1 Measuring CPU time and wall time

The CPU time of a tool must be measured and limited accurately, including the CPU time of all (transitive) child processes that the tool started. The wall time (i.e., the elapsed time between start and end of the tool execution) must be measured without being affected by changes to the system clock, e.g., due to daylight-savings time or due to time adjustments in the background that are for example caused by NTP services.

2.1.2 Measuring peak memory consumption

For benchmarking, we are interested in the peak resource consumption of a process, i.e., the smallest amount of resources with which the tool could successfully be executed with the same result. Thus, the memory usage of a process is defined as the peak size of all memory pages that occupy some system resources. This means, for example, that we should not measure and limit the size of the address space of a process, because it may be much larger than the actual memory usage. This might happen due to memory-mapped files or due to allocated but unused memory pages (which do not actually take up resources, because the Linux kernel lazily allocates physical memory for a process only when a virtual memory page is first written to, not when it is allocated). The size of the heap can also not be used as memory measure because it does not include the stack, and the so-called resident set of a process (the memory that is currently kept in RAM) cannot be used because it does not include pages that are in use but have been swapped out.

If a tool spawns several processes, these can use shared memory, such that the total memory usage of a group of processes is less than the sum of their individual memory usages. Shared memory occupies system resources only once and thus needs to be counted only once by the benchmarking framework.

Explicitly setting a limit for memory usage is important and should always be done, because otherwise the amount of memory available to the tool is the amount of free memory in the system, which varies over time and depends on lots of external factors, preventing repeatable results.

2.2 Terminate processes reliably

If a resource limit is violated, it is necessary to reliably terminate the tool including all of its child processes. Even if the tool terminates itself, the benchmarking environment needs to ensure that all child processes are also terminated. Otherwise a child process could keep running and occupy CPU and memory resources, which might influence later benchmarks on the same machine.

2.3 Assign cores deliberately

Special care is necessary for the selection of CPU cores that are assigned to one tool execution. For the scheduler of the OS, a core is a processing unit that allows execution of one thread independently of what happens on a different core. However, on the hardware level cores are usually not fully independent because of shared hardware resources. In this case, the performance of a core and thus the CPU-time and wall-time measurements of a tool execution are influenced by the actions of threads running on other cores, and the performance impact depends not only on the characteristics of the machine's hardware, but also on the type of operations performed by all these threads, and on the timing of their operations relative to each other. For example, if all threads are heavily accessing the memory at the same time, a larger influence is to be expected than if the threads happen to access the memory at different times. Because we cannot guarantee an upper bound on the size of the performance influence and because the influence is nondeterministic, we need to avoid such performance influences as far as possible in order to achieve accurate measurements.

If the benchmarked tool is concurrent and should be executed using several cores, avoiding performance influences between the tool's *own* threads and processes is specific to the tool and thus the responsibility of the tool itself or the user. For performance influences on the tool from *other* processes, the most reliable way to avoid them would be to execute only one instance of the benchmarked tool at the same time, and to ensure that no other processes outside the control of the benchmarking environment are active. However, this might not always be possible, for example, due to machines being shared with other users, or due to the amount of benchmarking work being so large that parallel executions are required.

In such cases, the benchmarking environment should assign a fixed set of CPU cores to each (parallel) tool execution and not let the scheduler of the OS assign cores dynamically, in order to prevent additional performance influences from processes being moved between cores. The benchmarking environment needs to compute the mapping of CPU cores per run such that the CPU cores for one run are as close to each other as possible and the sets of cores for separate runs are as independent as possible on the hardware level,[7] and thus the performance impact is minimized. Users need to know about the characteristics of their benchmarking machine and what kind of performance influences they need to expect, in order to properly decide how many parallel executions are acceptable for a given benchmark.

For example, we would usually consider it acceptable to have parallel tool executions on different CPUs that each have their own local memory, whereas it would usually considered to be *not* acceptable to have parallel tool executions on different virtual cores of the same physical core (just like there should never be two simultaneous tool executions sharing one virtual core). More information on how hardware characteristics can affect performance is given in Sect. 4. In any case, the used resource allocation should be documented and made available together with the benchmark results.

2.4 Respect nonuniform memory access

Systems with several CPUs often have an architecture with nonuniform memory access (NUMA), which also needs to be considered by a benchmarking environment. In a NUMA architecture, a single CPU or a group of CPUs can access parts of the system memory locally, i.e., directly, while other parts of the system memory are remote, i.e., they can only be accessed indirectly via another CPU, which is slower. The effect is that once a process has to access remote memory, this leads to a performance decrease depending on the load of the inter-CPU connection and the other CPU. Hence, a single tool execution should be bound to memory that is local to its assigned CPU cores, in order to avoid nondeterministic delays due to remote memory access.

2.5 Avoid swapping

Swapping out memory must be avoided during benchmarking, because it may degrade performance in a nondeterministic way. This is especially true for the benchmarked process(es), but even swapping of an unrelated process can negatively affect the benchmarking, if the benchmarked process has to wait for more free memory to become available. Absolutely preventing swapping can typically only be done by the system administrator by turning off all available swap space. In theory, it is not even enough to ensure that the OS, the benchmarking environment, and the benchmarked processes all fit into the available memory, because the OS can decide to start swapping even if there is still memory available, for example, if it decides to use some memory as cache for physical disks. However, for benchmarking CPU-bound tools, with high CPU and memory usage, and next to no I/O, this is unlikely to happen with a modern OS. Thus, the main duty of the benchmarking environment is to ensure that there is no overbooking of memory, and that memory limits are enforced effectively. It is also helpful if the benchmarking environment monitors swap usage during benchmarking and warns the user of any swapping.

2.6 Isolate individual runs

If several tool executions are executed in parallel, and to some extent even if they are executed sequentially, the different instances of the benchmarked tool(s) can interfere with each other, which could influence the performance and/or change the results.

One common reason for mutual interference are write accesses to shared files in the temp directory and in the home directory. For example, if a tool uses a temporary file with a fixed name, a cache directory, or configuration files in the home directory, parallel instances may interfere with each other nondeterministically. Even if runs are executed strictly sequentially, left-over files from previous runs could influence later runs if the tool reads these files, and prevent repeatability of experiments (because results then depend on the order of executing the runs).

Another reason for mutual interference of parallel runs are signals like `SIGKILL` or `SIGTERM`, if they get sent to processes that belong to a different tool instance. This may happen inadvertently, for example in a well-meaning cleanup script that tries to terminate child processes of a tool with the command `killall`.

The benchmarking environment should isolate the benchmarked processes to prevent such interference.

3 Limitations of existing methods

Some of the methods that are available on Linux systems for measuring resource consumption and for enforcing resource limits of processes have several problems that make them unsuitable for benchmarking, especially if child processes are involved. Any benchmarking environment needs to be aware of these limitations and avoid using naive methods for resource measurements. However, later on in Sect. 8.2 we will see that indeed a number of existing benchmarking tools use the methods described in this section and thus do not ensure reliable results.

[7] i.e., with high cohesion and loose coupling

3.1 Measuring resources may fail

3.1.1 Measuring CPU time and wall time

Measuring wall time is sometimes done by reading the system clock at start and end of a run and calculating the difference. However, because the system clock can jump due to time adjustments and even change its pace, it is important to use a time source that is guaranteed to be strictly monotonic and of constant rate. Many operating systems and programming languages offer such a time source with high precision specifically for benchmarking.

Measuring CPU time of the main process of a tool, for example using the tool `time` or a variant of the system call `wait` (which returns the CPU time after the given process terminated), does not reliably include the CPU time of child processes that were spawned by the main process. The Linux kernel only adds the CPU time used by child processes to that of the parent process *after* the child process has terminated *and* the parent process waited for the child's termination with a variant of the system call `wait`. If the child process has not yet terminated or the parent did not explicitly wait for its termination, the CPU time of the child is lost. This is a typical situation that might happen, for example, if a verifier starts an SMT solver as a child process and communicates with the solver via `stdin` and `stdout`. When the analysis finishes, the verifier would terminate the solver process, but usually would not bother to wait for its termination. A tool that runs different analyses in parallel child processes would also typically terminate as soon as the first analysis returns a valid result, without waiting for the other analyses' termination.[8] In these cases, a large share of the total CPU time is spent by child processes but not included in the measurement.

3.1.2 Measuring peak memory consumption

Some measurement tools only provide a view on the current memory usage of individual processes, but we need to measure the *peak* usage of a *group* of processes. Calculating the peak usage by periodically sampling the memory usage and reporting the maximum is inaccurate, because it might miss peaks of memory usage. If the benchmarked process started child processes, one has to recursively iterate over all child processes and calculate the total memory usage. This contains several race conditions that can also lead to invalid measurements, for example, if a child process terminates before its memory usage could be read. In situations where several processes share memory pages (e.g., because each of them loaded the same library, or because they communicate via shared memory), we cannot simply sum up the memory usage of all processes. Thus, without keeping track of every memory page of each process, manually filtering out pages that do not occupy resources because of lazy allocation, and counting each remaining page exactly once, the calculated value for memory usage is invalid. If all this is done with a high sampling frequency (to not miss short peaks of memory usage), we risk that the benchmarked process is being slowed down by the increased CPU usage.

3.2 Enforcing limits may fail

For setting resource limits, some users apply the tool `ulimit`, which uses the system call `setrlimit`. A limit can be specified for CPU time as well as for memory, and the limited process is forcefully terminated by the kernel if one of these limits is violated. However, similar to measuring time with the system call `wait`, limits imposed with this method affect only individual processes, i.e., a tool that starts n child processes could use n times more memory and CPU time than allowed. Limiting memory is especially problematic because either the size of the address space or the size of the data segment (the heap) can be limited, which do not necessarily correspond to the actual memory usage of the process, as described in Sect. 2.1.2. Limiting the resident-set size (RSS) is no longer supported.[9] Furthermore, if such a limit is violated, the kernel terminates only the one violating process, which might not be the main process of the tool. In this case it depends on the tool itself how such a situation is handled: it might terminate itself, or crash, or even continuously respawn the terminated child process and continue. Thus, this method is not reliable.

It is possible to use a self-implemented limit enforcement with a process that samples CPU time and memory usage of a tool with all its child processes, terminating all processes if a limit is exceeded, but this is inaccurate and prone to the same race conditions, as described above for memory measurement.

3.3 Termination of processes may fail

In order to terminate a tool and all its child processes, one could try to (transitively) enumerate all its child processes and terminate each of them. However, finding and terminating all child processes of a process may not work reliably for two reasons. First, a process might start child processes faster than the benchmarking environment is able to terminate them. While this is known as a malicious technique ("fork bomb"), it may also happen accidentally, for example due to a flawed logic for restarting crashed child processes of a

[8] We experienced this when organizing SV-COMP'13, for a portfolio-based verifier. Initial CPU-time measurements were significantly too low, which was only discovered by chance. The verifier had to be patched to wait for its subprocesses and the benchmarks had to be rerun.

[9] http://man7.org/linux/man-pages/man2/setrlimit.2.html

tool. The benchmarking environment should guard against this, otherwise the machine might become unusable. Second, it is possible to "detach" child processes such that they are no longer recognizable as child processes of the process that started them. This is commonly used for starting long-running daemons that should not retain any connection to the user that started them, but might also happen incidentally if a parent process is terminated before the child process. In this case, an incomplete benchmarking framework could miss to terminate child processes.

The *process groups* of the POSIX standard (established with the system call `setpgid`[10]) are not reliable for tracking child processes. A process is free to change its process group, and tools using child processes often use this feature.

3.4 Hardware allocation may be ineffective

There are two mistakes that can be made when attempting to assign a specific set of CPU cores to a benchmark run. First, the set of CPU cores for a process can be specified with the tool `taskset` or alternatively by the system call `sched_setaffinity`. However, a process can change this setting freely for itself, and does not need to follow predefined core restrictions. Thus, this is not a reliable way to enforce a restriction of the CPU cores of a process.

Second, the numbering system of the Linux kernel for CPU cores is complex and not consistent across machines of different architectures (not even with the same kernel version). Any "naive" algorithm for assigning cores to tool executions (e.g., allocating cores with consecutive ids), will fail to produce a meaningful core assignment, and give sub-optimal performance, on at least some machines.

The Linux kernel assigns numeric ids for CPUs (named *physical [package] id*), for physical cores (named *core id*), and for virtual cores (named *processor id* or *CPU id*). The latter is used for assigning cores to processes. Additionally, there is a numeric id for NUMA memory regions (named *node id*), which is used for restricting the allowed memory regions of a process. There is no consistent scheme how these ids are assigned by the kernel. In particular, we have found the following intuitive assumptions to be *invalid*:

- CPUs, virtual cores, and NUMA regions are always numbered consistently (instead, the virtual cores or NUMA regions may be numbered in a different order than the CPUs).
- The core id is assigned consecutively (instead, there may be gaps).
- Virtual cores that belong to the same physical core have processor ids that are as far apart as possible.
- Virtual cores that belong to the same physical core have processor ids that are as close as possible (i.e., consecutive).
- Virtual cores that belong to the same physical core have the same core id.
- Virtual cores that belong to different physical cores have different core ids (instead, on systems with several NUMA regions per CPU, there are several physical cores on each CPU with the same core id).
- The tuple (physical [package] id, core id) uniquely identifies physical cores of the system.

For several of these invalid assumptions, a violation can be seen in Figs. 12 or 13 in Appendix A.

Therefore, we must not rely on any assumption about the numbering system and only use the explicit CPU topology information given by the kernel. The authoritative source for this information is the `/sys/devices/system/cpu/` directory tree. Note that the file `/proc/cpuinfo` neither contains the node id for NUMA regions nor the information which virtual cores share the same hardware, and thus cannot be used to compute a meaningful core assignment for benchmarks.

3.5 Isolation of runs may be incomplete

Processes can be isolated from external signals by executing each parallel tool execution under a separate user account. This also helps to avoid influences from existing files and caches in the user's home directory, but it does not allow separating the temporary directory for each run, and thus parallel runs can still influence each other if they use temporary files with hard-coded names. Many tools allow using the environment variable `TMPDIR` for specifying a directory that is used instead of `/tmp`, but not all tools support this option. For example, the Java VM (both Oracle and OpenJDK) ignores this variable.

4 Impact of hardware characteristics on parallel tool executions

Certain hardware characteristics of today's machines can influence the performance of tools that are executed in parallel on one machine. Because parallel executions, i.e., starting several independent instances of the benchmarked tool at the same time, are sometimes necessary in order to reduce the time that is necessary for performing experiments with many tool executions, it is important to understand the sources of undesired performance influences and how to minimize their impact. In this section, we present an overview of important hardware characteristics and highlight the effects they can have on parallel tool executions based on experimental

[10] http://man7.org/linux/man-pages/man2/setpgrp.2.html

results. Note that while similar effects occur for concurrent tools, i.e., tools that employ several threads or processes, we consider here only the effects between parallel tool executions and not those between the processes and threads inside a single tool execution (cf. Sect. 2.3).

4.1 Overview of hardware characteristics

In today's machines, CPU cores can typically be roughly organized in a hierarchy with three layers:[11] A machine may have multiple *CPUs* (also named sockets or [physical] packages), each CPU may have multiple *physical cores*, and each physical core may have multiple *virtual cores* (also named [hardware] threads or processors). The virtual cores of one physical core may share some execution units, level-1, and level-2 caches (e.g., in case of hyperthreading). The physical cores of one CPU may share a level-3 cache and the connection to the RAM. The CPUs of one machine may share the connection to the RAM and to I/O. If not all cores on the system share the same connection to the RAM, the system is said to have a NUMA architecture. Appendix A shows examples of two hardware architectures.

The "closer" two cores are, i.e., the more hardware they share, the larger can be the mutual performance influence of two threads running on them in parallel. The largest influence is to be expected by hyperthreading (several virtual cores sharing execution units) or the "module" concept of AMD CPUs (several virtual cores sharing level-1 and level-2 caches). Even if we use separate physical cores for each tool execution, there can be further nondeterministic influences on performance. Modern CPUs often adjust their frequency by several hundred MHz depending on how many of their cores are currently used, with the CPU running faster when less cores are used (this is commonly called "Turbo Boost" or "Turbo Core"). For memory-intensive programs, the influence by other processes running on different physical cores that share the same level-3 cache and the connection to the RAM (and thus compete for memory bandwidth) can also be significant.

On the other hand, the closer two cores are, the faster they can typically communicate with each other. Thus, we can reduce performance impact from communication by allocating cores that are close together to each tool execution.

4.2 Experiment setup

To show that these characteristics of the benchmarking machine can significantly influence the performance, and thus have a negative influence on benchmarking if not handled appropriately, we conducted several experiments on a machine with two Intel Xeon E5-2650 v2 CPUs and 68 GB of RAM per CPU. The CPUs (each with eight physical cores) support hyperthreading (with two virtual cores per physical core) and Turbo Boost (base frequency is 2.6 GHz, with up to 3.4 GHz if only one core is used). The machine has a NUMA architecture. A graphical overview of this system can be seen in Fig. 13 of Appendix A.

As an example for a benchmarked tool we took the verifier CPACHECKER[12] in revision 17829 from the project repository[13] with its configuration for predicate analysis. The machine was running a 64-bit Ubuntu 14.04 with Linux 3.13 and OpenJDK 1.7 as Java virtual machine. As benchmark set we used 4011 C programs from SV-COMP'15 [5] (excluding categories not supported by this configuration of CPACHECKER). In order to measure only effects resulting from concurrency between parallel tool executions and avoid effects from concurrency inside each tool execution, we restricted each tool execution to one virtual core of the machine. We also limited each tool execution to 4.0 GB of memory and 900 s of CPU time. Except were noted, Turbo Boost was disabled such that all cores were running at 2.6 GHz. Apart from our tool executions, the machine was completely unused. All experiments were conducted with BENCHEXEC 1.2. We show the accumulated CPU time for a subset of 2414 programs that could be solved by CPACHECKER within the time and memory limit on the given machine. (Including timeouts in the accumulated values would skew the results.) Time results were rounded to two significant digits. Tables with the full results and the raw data are available on our supplementary web page.[14]

Note that the actual performance impact of certain hardware features will differ according to the characteristics of the benchmarked tool and the benchmarking machine. For example, a tool that uses only little memory but fully utilizes its CPU core(s) will be influenced more by hyperthreading than by nonlocal memory, whereas it might be the other way around for a tool that relies more on memory accesses. In particular, the results that are shown here for CPACHECKER and our machine are not generalizable and show only that there *is* such an impact. Because the quantitative amount of the impact is not predictable and might be nondeterministic, it is important to avoid these problems in order to guarantee reliable benchmarking.

[11] Systems can be even more complex and have more layers. However, the hierarchy presented here captures the facts that are most important for the performance of software from our target domain. Thus, we use this abstracted definition and nomenclature.

[12] https://cpachecker.sosy-lab.org

[13] https://svn.sosy-lab.org/software/cpachecker/trunk

[14] https://www.sosy-lab.org/research/benchmarking#benchmarks

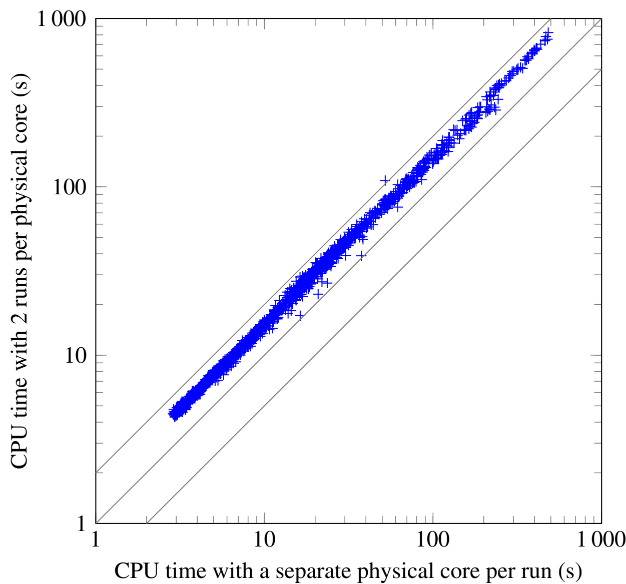

Fig. 2 Scatter plot showing the influence of hyperthreading for 2414 runs of CPAchecker: the data points above the diagonal show a performance decrease due to an inappropriate assignment of CPU cores

4.3 Impact of hyperthreading

To show the impact of hyperthreading, which is also named simultaneous multithreading, we executed the verifier twice in parallel on one CPU of our machine. In one part of the experiment, we assigned each of the two parallel tool executions to one virtual core from separate physical cores of the same CPU. In a second part of the experiment, we assigned each of the two parallel tool executions to one virtual core from the same physical core, such that both runs had to share the hardware resources of one physical core. A scatter plot with the results is shown in Fig. 2. For the 2414 solved programs from the benchmark set, 16 h of CPU time were necessary using two separate physical cores and 25 h of CPU time were necessary using the same physical core, an increase of 53% caused by the inappropriate core assignment. This shows that hyperthreading can have a significant negative impact, and parallel tool executions should not be scheduled on the same physical core.

4.4 Impact of shared memory bandwidth and caches

To show the impact of a shared memory connection and a shared level-3 cache for multiple physical cores, we experimented with 1 to 8 parallel tool executions on the same CPU (each on its own physical core), i.e., with 1 to 8 used physical cores that share a common level-3 cache and the memory bandwidth. The second virtual core of every physical core was unused, and Turbo Boost was disabled. A plot with the results is shown in Fig. 3 (shaded blue bars). For the 2414 solved programs from the benchmark set, 16 h of CPU time were necessary if only one physical core was used, whereas 20 h of CPU time were necessary with eight physical cores used; the increase of 22% is caused by the contention on cache and memory accesses. The linear-regression line in the plot shows that the used CPU time scales linearly with the amount of used cores. This can be explained by the CPU dynamically allocating equal shares of its memory bandwidth and level-3 cache to each of those cores that are actively used.

This experiment shows that parallel tool executions on the same CPU can influence the measured CPU time significantly, and thus need to be avoided for maximal precision and accuracy of benchmarking results. However, because the time that users have to wait for the results of experiments with many tool executions can be decreased drastically by using parallel tool executions, it might in practice sometimes be necessary to compromise and use at least a few parallel tool executions during benchmarking. How many parallel instances are acceptable depends on the used hardware and the user's requirements on precision and accuracy and cannot be answered in general. Note that by utilizing only a subset of the cores of a CPU the size of the undesired influence on CPU time can be reduced while keeping most of the time savings compared to no parallelization at all.

4.5 Impact of Turbo Boost

To show the impact of Turbo Boost, we executed benchmarks with the same setup as in Sect. 4.4, but now with Turbo Boost enabled. This means that the CPU uses a higher frequency depending on the current load of its cores. Without Turbo Boost, a used core of this CPU always runs at 2.6 GHz. With Turbo Boost, a single used core of this CPU can run at 3.4 GHz, if the CPU is otherwise idle, and even if all eight cores are used, they can still run at 3.0 GHz. The results are also shown in Fig. 3 (red bars). As expected, due to the higher frequency, the CPU time is lower than with Turbo Boost disabled, and the more physical cores are used in parallel, the higher the used CPU time becomes. The latter effect is larger if Turbo Boost is enabled than if Turbo Boost is disabled, because in addition to the contention of cache and memory bandwidth, there is now the additional performance influence of the varying CPU frequency. Instead of increasing by 22%, the used CPU time now increases by 39% (from 13 to 18 h) if using eight instead of one physical core. Thus, a dynamic scaling of the CPU frequency should be disabled if multiple tool executions run in parallel on a CPU.

4.6 Impact of NUMA

To show the impact of NUMA, we executed 16 instances of the verifier in parallel, one instance per physical core of the two CPUs of our machine. In one part of the experi-

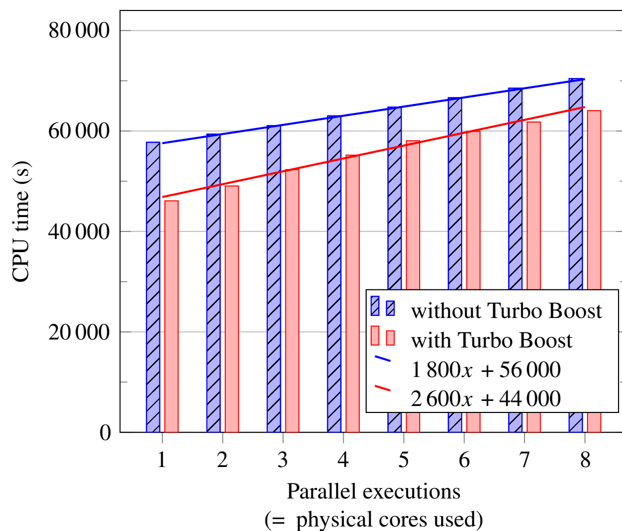

Fig. 3 Plot showing the influence of Turbo Boost, shared level-3 cache, and shared memory bandwidth for 2414 runs of CPACHECKER

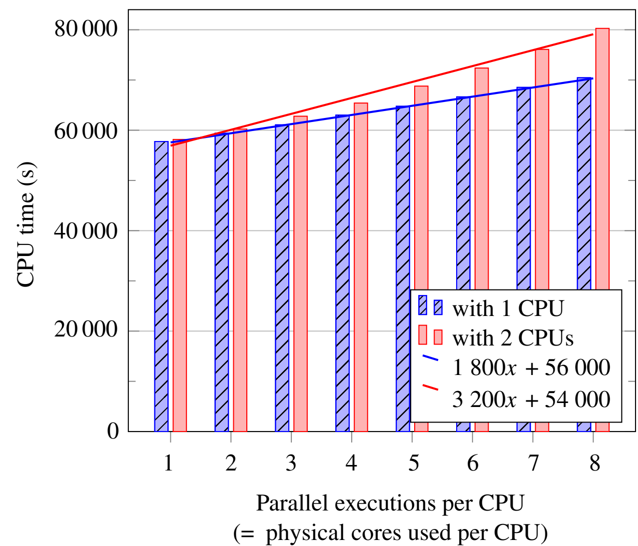

Fig. 4 Plot showing the influence of using multiple CPUs for 2414 runs of CPACHECKER

ment, we assigned memory to each tool execution that was local to the CPU that the tool was executed on. In a second part of the experiment, we deliberately forced each of the 16 tool executions to use only memory from the *other* CPU, such that all memory accesses were indirect. For the 2414 solved programs from the benchmark set, 23 h of CPU time were necessary using local memory and 24 h of CPU time were necessary using remote memory, an increase of 6.8% caused by the inappropriate memory assignment. While the performance impact in this case is not that large, there exists no reason to not ensure a proper memory assignment in the benchmarking environment and rule out this influence completely.

4.7 Impact of multiple CPUs

To show the impact of multiple CPUs used in parallel, we experimented with 1 to 16 parallel tool executions across all CPUs. For one part of the experiment, we used the same setup as in Sect. 4.4: using only one CPU and executing 1 to 8 parallel tool executions, each on a designated physical core (hence, the shaded blue bars in Fig. 4 match exactly the shaded blue bars in Fig. 3). For a second part of the experiment, we used both CPUs of the machine, executing 1 to 8 parallel tool executions *on each CPU*, i.e., with 2 to 16 parallel tool executions on the machine. A summary of the results is shown in Fig. 4. If only one physical core of the whole machine is used, 16 h of CPU time were necessary for the 2414 tool executions. This increases linearly to 20 h of CPU time if all eight cores of one CPU are used (shaded blue bars in the plot). If one physical core of each of the two CPUs is used (i.e., two parallel tool executions), the necessary CPU time is also 16 h: there is no significant difference to using one physical core of only one CPU. However, if more physical cores per CPU are used, using n physical cores of each of the two CPUs with $2n$ parallel tool executions (red bars) is slower than using n physical cores of only one of the CPUs. The maximal difference occurs for eight cores per CPU, which uses 22 h of CPU time compared to 20 h (an increase of 14%).

We also show a scatter plot in Fig. 5 with more detail on the last part of the experiment, i.e., for 8 used physical cores per CPU, respectively. It shows that for some of the 2414 runs, the performance was actually equal in both cases (the data points on the diagonal), whereas for others the CPU time is almost doubled (the data points close to the gray line for $y = 2x$).

This experiment shows that parallelization of tool executions across multiple CPUs at least sometimes needs to be treated similarly to parallelization of tool executions across multiple cores of the same CPU.

4.8 Investigation of impact of multiple CPUs

Because we did not expect the performance impact of using multiple CPUs with separate local memory for independent parallel tool executions, we tried to find out the source of this effect using additional experiments and to ensure that other possible performance influences were ruled out as far as possible. Unfortunately we did not find an explanation so far, and the respective machine is no longer available for further experiments, so the reason for the performance impact of multiple CPUs remains unknown to us. However, we document these experiments here in the hope that this will help others to investigate this issue in the future.

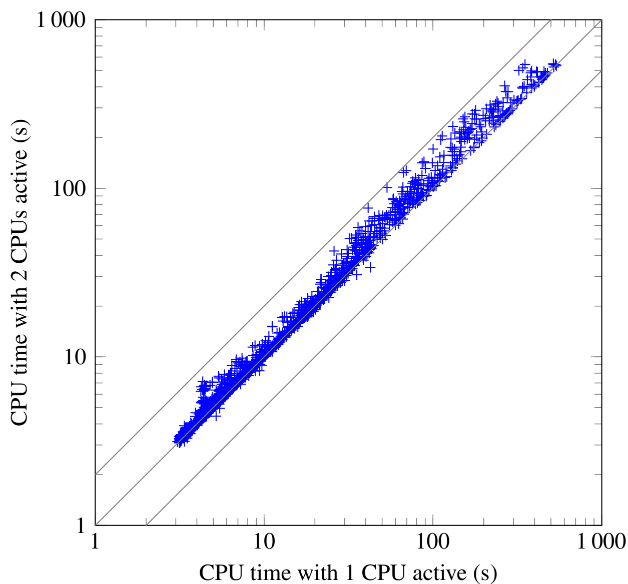

Fig. 5 Scatter plot showing the influence of using multiple CPUs for 2414 runs of CPACHECKER: the data points above the diagonal show a performance decrease due to using two CPUs in parallel instead of only one CPU

We repeated the complete experiment from Sect. 4.7 at least two times, and the performance variation across the three experiments was negligible compared to the difference between using one or two CPUs. We also tried to reproduce this effect on two other multi-CPU machines, both with AMD CPUs (four Opteron 6380 and eight Opteron 8356, respectively). In both cases there was no significant performance change for varying the number of used CPUs. So we were not able to reproduce the effect on the AMD CPUs. Unfortunately, we did not have access to other multi-CPU machines with Intel CPUs.

On our machine with Intel CPUs, the effect of decreased performance if both CPUs are active did not occur in experiments with a tool that uses mainly the CPU and not much memory. To measure this, we experimented with the same setup for the tool `bc` (an arbitrary-precision calculator) and let it compute π with 1000 to 19 890 digits using the formula $\arctan(1) \times 4$. This takes between 0.43 and 880 s on our machine and uses less than 970 kB of memory.

Both CPUs in the machines with the two Intel CPUs have their own directly connected memory (visible in Fig. 13 of Appendix A), and we made sure that all tool executions use only memory belonging to the same CPU as their CPU core(s) to avoid performance influences from NUMA. We also made sure that even the start of each tool execution already occurred on its assigned core, such that migration between cores, or even between CPUs, was avoided. There are no hardware caches that are shared between the CPUs in this system, and no swap space. There is also no correlation between the fact that a tool execution experienced a slow-down if both CPUs were active and the specific CPU that the tool was executed on. For both CPUs, there were tool executions with unaffected performance as well as tool executions with significantly increased CPU time.

The I/O (which would be a shared bottleneck) was kept to a minimum during benchmarking. Apart from loading the Java virtual machine and the verifier, each run used I/O only for loading the input file and for writing a log file with the output of the verifier (output of each tool execution was 15 kB on average and always less than 750 kB). Switching between a local SSD and a file server connected via network for all I/O except loading the JVM did not influence the performance, indicating that I/O indeed was not a bottleneck. We also verified that no additional temporary files were used. The tool executions were independent and the processes did not communicate with each other. Enabling Turbo Boost decreased the necessary CPU time in general and made the effect slightly larger. Upgrading the Linux kernel from version 3.13 to 3.19 improved the performance in general by approximately 1 to 2%, but did not change the effect of decreased performance for using multiple CPUs. The machine is kept in an air-conditioned server room and did not throttle due to overheating during benchmarking.

For further investigation, we collected some statistics about the CPUs using the perf framework of the Linux kernel[15] while executing the tool. (These statistics are collected using internal hardware counters of the CPU and do not affect performance.) The result was that the number of context switches, CPU-migrations of threads, page faults, executed instructions and branches, and branch misses for one execution of the whole benchmark set were nearly the same, regardless of whether one or two CPUs were used, and thus gave no indication where the performance impact could come from.

One possible reason could be that Linux keeps only one global file-system cache that is shared for all processes from all CPUs. This means that, after a file has been loaded into the file-system cache, accessing the content of this file will be somewhat faster from one CPU than from the other CPU(s) due to NUMA. This behavior cannot be disabled. To avoid this effect, we used two separate on-disk copies of the benchmarked tool, and each copy was used for all runs executing on one specific CPU. Compared to using one on-disk copy of the benchmarked tool, this did not change the performance. While there was still only a single install of the JVM used for all tool executions, we can assume that the single copy of the JVM in the file-system cache of the Linux kernel is not the reason for this effect. After all, the experiment from Sect. 4.6 showed a smaller effect even if *all* memory accesses were indirect, not only those to the JVM files in the file-system cache.

[15] https://perf.wiki.kernel.org

5 State-of-the-art benchmarking with cgroups and containers

We listed aspects that are mandatory for reliable benchmarking, and explained flaws of existing methods. In the following, we present two technologies that can be used to avoid these pitfalls.

5.1 Introducing cgroups for benchmarking

Control groups (cgroups) are a feature of the Linux kernel for managing processes and their resource usage, which is available since 2007 [28]. Differently from all other interfaces for these problems, cgroups provide mechanisms for managing groups of processes and their resources in an atomic and race-free manner, and are not limited to single processes. All running processes of a system are grouped in a hierarchical tree of cgroups, and most actions affect all processes within a specific cgroup. Cgroups can be created dynamically and processes can be moved between them. There exists a set of so-called *controllers* in the kernel, each of which controls and measures the consumption of a specific resource by the processes within each cgroup. For example, there are controllers for measuring and limiting CPU time, memory consumption, and I/O bandwidth.

The cgroups hierarchy is made accessible to programs and users as a directory tree in a virtual file system, which is typically mounted at `/sys/fs/cgroups`. Usual file-system operations can be used to read and manipulate the cgroup hierarchy and to read resource measurements and configure limits for each of the controllers (via specific files in each cgroup directory). Thus, it is easy to use cgroups from any kind of tool, including shell scripts. Alternatively, one can use a library such as `libcg`,[16] which provides an API for accessing and manipulating the cgroup hierarchy. Settings for file permission and ownership can be used to fine-tune who is able to manipulate the cgroup hierarchy.

When a new process is started, it inherits the current cgroup from its parent process. The only way to change the cgroup of a process is direct access to the cgroup virtual file system, which can be prevented using basic file-system permissions. Any other action of the process, whether changing the POSIX process group, detaching from its parent, etc., will not change the cgroup. Thus, cgroups can be used to reliably track the set of (transitive) child processes of any given process by putting this process into its own cgroup. We refer to the manual for details.[17]

The following cgroup controllers are relevant for reliable benchmarking:

`cpuacct` measures, for each cgroup, the accumulated CPU time that is consumed by all processes of the cgroup. A time limit cannot be defined, but can be implemented in the benchmarking environment by periodically checking the accumulated time.

`cpuset` supports restricting the processes in a cgroup to a subset of the available CPU cores. On systems with more than one CPU and NUMA, it allows restricting the processes to specific parts of the physical memory. These restrictions are applied additionally to those set with `sched_setaffinity`, such that changes to the latter will not affect restrictions made via cgroups.

`freezer` supports freezing all processes of a cgroup in a single operation. This can be used for reliable termination of a group of processes by freezing them first, sending the kill signal to all of them, and afterward unfreezing ("thawing") them. This way the processes do not have the chance to start other processes because between the time the first and the last process receive the kill signal, none of them can execute anything.

`memory` supports, for each cgroup, restricting maximum memory usage of all processes together in the cgroup, and measuring current and peak memory consumption. If the defined memory limit is reached by the processes in a cgroup, the kernel first tries to free some internal caches that it holds for these processes (for example disk caches), and then terminates at least one process. Alternatively, instead of terminating processes, the kernel can send an event to a registered process, which the benchmarking framework can use to terminate all processes within the cgroup. The kernel counts only actually used pages toward the memory usage, and because the accounting is done per memory page, shared memory is handled correctly (every page that the processes use is counted exactly once).
The `memory` controller supports two limits for memory usage, one on the amount of physical memory that the processes can use, and one on the amount of physical memory plus swap memory. If the system has swap memory, both limits need to be set to the same value for reliable benchmarking. If only the former limit is set to a specific value, the processes could use so much memory plus all of the available swap memory (and the kernel would automatically start swapping out the processes if the limit on physical memory is reached). Similarly, for reading the peak memory consumption, the value of physical memory plus swap memory should be used. Sometimes, the current memory consumption of a cgroup is not zero even after all processes of the cgroup have been terminated, if the kernel decided to still keep some pages of these processes in its disk cache. To avoid influencing the measurements of later tool executions

[16] http://libcg.sourceforge.net

[17] https://www.kernel.org/doc/Documentation/cgroup-v1

by this, a cgroup should be used only for a single run and deleted afterward, with a new tool execution getting a new, fresh cgroup.[18]

As described in Sect. 3.4, the Linux kernel does not use a consistent scheme for assigning processor ids to virtual cores and node ids to memory regions, which are the ids used by the `cpuset` controller. Information about the hardware topology and the relations of CPU cores to each other and to memory regions needs to be read from the directory tree `/sys/devices/system/cpu/`.[19] There, one can find a directory named `cpu`<*i*> for each virtual core *i*, and inside each such directory, the following information is present: the symlinks named `node`<*j*> point to the NUMA region(s) of this virtual core, `topology/physical_package_id` contains the physical id of this virtual core, `topology/core_id` contains the core id of this virtual core, `topology/core_siblings_list` contains the virtual cores of the same CPU as this virtual core, and `topology/thread_siblings_list` contains the virtual cores of the same physical core as this virtual core.

5.2 Benchmarking containers based on namespaces

Container is a common name for an instance of OS-level virtualization on Linux. Contrary to virtual machines, there is no virtual hardware simulated for a container, and a container does not have its own kernel running. Instead, the applications in a container run directly on the same kernel as applications outside the container, without any additional layers that would reduce performance. However, the kernel provides a limited view of the system to processes inside a container, such that these processes are restricted in what they can do with regard to the system outside their container. Containers can be used to execute a single application in isolation. A well-known framework for creating containers in Linux is Docker.

The key technology behind containers are *namespaces*,[20] which are a feature of the Linux kernel for providing individual processes or groups of processes with a different view on certain system resources compared to other processes. There exist different kinds of namespaces, which can be used individually or in combination, each responsible for isolating some specific system resources. For example, assigning a different network namespace to a process will change which network interfaces, IP addresses, etc., the process sees and is able to use. Assigning a different namespace for process ids (PIDs) will change which processes can be seen, and which PIDs they seem to have.

Using namespaces, we can create benchmarking containers that prevent any communication or interference with processes outside the container. We need only a modern ($\geq$3.8) Linux kernel for this; no other software is necessary. The benchmarked tool will typically not notice that it is executed in such a container. The performance of executing a process within a separate namespace (i.e., inside a container) is comparable to executing it directly in the initial namespace, because the process still interacts directly with the same kernel and there are no additional layers like in hardware-virtualization solutions.

Since Linux 3.8 (released February 2013) it is possible to create and configure namespaces as a regular user without additional permissions. At a first glance it may seem that creating and joining a namespace can give a process more permissions than it previously had (such as changing network configuration or file-system mounts), but all these new permissions are only valid inside the namespace, and none of these actions affect the system outside of this namespace.

The Linux kernel provides the following namespaces that are relevant for reliable benchmarking:

`mount` namespaces allow changing how the file-system layout appears to the processes inside the namespace. Existing directories can be mounted into a different place using "bind" mounts (similar to symbolic links), and mount points can be set read-only. Unprivileged users cannot create new mounts of most file systems, even if they are in a mount namespace, but for a few special file systems this is allowed. For example, RAM disks can be mounted by everyone in a mount namespace.

`ipc` namespaces provide separation of several different forms of interprocess communication (IPC), such as POSIX message queues.

`net` namespaces isolate the available network interfaces and their configuration, e.g., their IP addresses. By default, a new network namespace has only a loopback interface with the IP addresses `127.0.0.1` (IPv4) and `::1` (IPv6) and thus, processes in such a namespace have no access to external network communication, but can still use the loopback interface for communication between the processes within the same namespace (note that the loopback interfaces of different network namespaces are different: a loopback interface cannot be used for communication with processes of separate network namespaces).

`pid` namespaces, which can be nested, provide a separate range of process IDs (PIDs), such that a process can have different PIDs, one in each (nested) namespace that it is part of. Furthermore, the PID namespace also affects mounts of the `/proc` file system, which is the place where the Linux kernel

[18] Or clear the caches with `drop_caches`.

[19] Or use a library that does this reliably.

[20] http://man7.org/linux/man-pages/man7/namespaces.7.html

provides information about the currently running processes. If a new `/proc` file system is mounted in a PID namespace, it will list only those processes that are part of this namespace. Thus, a PID namespace can be used to restrict which other processes a process can see: only such processes that are in the same PID namespace are visible. This prevents, for example, sending signals to processes and killing them, if they are not visible in the current namespace. Additionally, because the first process in every new PID namespace has a special role (like the `init` process of the whole system), the kernel automatically terminates all other processes in the namespace when this first process terminates. This can be used for a reliable cleanup of all child processes.

`user` namespaces provide a mapping of user and group ids, such that a certain user id inside a namespace appears as a different id outside of it. Creating a user namespace is necessary for regular (non-`root`) users in order to create any other kind of namespace.

For each kind of namespace, an initial namespace is created at boot. These are the namespaces that are used for all processes on most systems. Processes can create new namespaces and move between them, but for the latter they need to get a reference to the target namespace, which can be prevented by using the PID namespace. Thus, it is possible to prevent processes from escaping back into an existing namespace such as the initial namespace. Furthermore, all namespace features are implemented in a way such that it is not possible to escape the restrictions of a namespace by creating and joining a fresh namespace: a freshly created namespace will not allow viewing or manipulating more system resources than its parent.

With mount namespaces we can customize the file-system layout. We can, for example, provide an own private `/tmp` directory for each tool execution by binding a freshly created directory to `/tmp` in the container of the tool execution. This avoids any interference between tool executions due to files in `/tmp`, and the same solution can also be applied for all other directories whose actual content should be invisible in the container. In order to ensure that no files created by a tool execution are left over in other directories (and possibly influence later tool executions), we can set all other mount points to read-only in the container. However, this can be inconvenient, for example, if the tool should produce some output file, or if it expects to be able to write in some fixed directory (like the home directory). The solution for this is to use an overlay file system.

Overlay file systems[21] use two existing directories, and appear to layer one directory over the other. Looking at the overlay file system, a file in the upper layer shadows a file with the same name in the lower layer. All writes done to the overlay file system go to the upper layer, never to the lower layer, which is guaranteed to remain unchanged. We can use this to present a directory hierarchy to the benchmarked tool that is initially equal to the directory hierarchy of the system and appears to be writable as usual, but all write accesses are rerouted to a temporary directory and do not affect the system directory. To do so we mount an overlay file system with the regular directory hierarchy of the system as lower layer and an empty temporary directory as upper layer.

An overlay file system is available in the Linux kernel since version 3.18, and on Ubuntu it can be used by regular users inside a mount namespace. On other distributions or older kernels we have to fall back to either read-only mounts or to giving direct write access to at least some directories of the file system.

6 BenchExec: a framework for reliable benchmarking

In the following, we describe our implementation BenchExec, a benchmarking framework for Linux that fulfills the requirements from Sect. 2 by using the techniques of cgroups and namespaces from Sect. 5. It is available on GitHub[22] as open source under the Apache 2.0 license.

BenchExec consists of two parts, both written in Python. The first part is `runexec`, responsible for benchmarking a single run of a given tool in isolation, including the reliable limitation and accurate measurement of resources, and encapsulates the use of cgroups and namespaces. This part is designed such that it is easy to use also from within other benchmarking frameworks. The second part is responsible for benchmarking a whole set of runs, i.e., executing one or more tools on a collection of input files by delegating each run to `runexec` and then aggregating the results. It consists of a program `benchexec` for the actual benchmarking and a program `table-generator` for postprocessing of the results.

6.1 System requirements

Full support for all features of BenchExec is available on Ubuntu with a Linux kernel of at least version 3.18 (default since Ubuntu 15.04). On older kernels or other distributions, the overlay file system cannot be used and the file-system layout inside the containers needs to be configured differently. On kernels older than Linux 3.8, BenchExec's use of containers needs to be disabled completely and thus isolation of runs will not be available, but other features of BenchExec

[21] https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

[22] https://github.com/sosy-lab/benchexec

(such has accurate resource measurements and limits, and sensible allocation of hardware resources) remain usable.

For the use of cgroups by BENCHEXEC, a few requirements are necessary that may demand assistance by the administrator of the benchmarking machine. Cgroups including the four controllers listed in Sect. 5.1 must be enabled on the system and the account for the benchmarking user needs the permissions to manipulate (a part of) the cgroup hierarchy. If the benchmarking machine has swap, swap accounting must be enabled for the `memory` controller. For enabling cgroups and giving permissions, we refer to standard Linux documentation. For more details on how to setup the prerequisites for BENCHEXEC, we refer to the respective chapter of the documentation.[23]

After these steps, no further root access is necessary and everything can be done with a normal user account. Thus, it is possible to use machines for benchmarking that are not under one's own administrative control. By creating a special cgroup for benchmarking and granting permissions only for this cgroup, it is also possible for the administrator to prevent the benchmarking user from interfering with other processes and to restrict the total amount of resources that the benchmarking user may use. For example, one can specify that a user may use only a specific subset of CPU cores and amount of memory for benchmarking, or partition the resources of shared machines among several users.

6.2 Benchmarking a single run

We define a *run* as a single execution of a tool, with the following input:

- the full command line, i.e., the path to the executable with all arguments, and optionally,
- the content supplied to the tool via `stdin`,
- the limits for CPU time, wall time, and memory, and
- the list of CPU cores and memory banks to use.

A run produces the following output:

- measurement values (e.g., CPU time, wall time, and peak memory consumption of the tool),
- the exit code of the main process,
- output written to `stdout` and `stderr` by the tool, and
- the files created or written by the tool.

The program `runexec` executes a tool with the given input, provides the output of the run, and ensures (using cgroups and namespaces) adherence to the specified resource limits, accurate measurement of the resource usage, isolation of the process with regard to network usage, signals, and file-system writes, and reliable cleanup of processes after execution (i.e., no process survives). The benchmarking containers created by `runexec` are illustrated in Fig. 6. If necessary, the benchmarking containers can be customized via additional options, e.g., with regard to the file-system layout (which directories are hidden in the container or made read-only etc.), or whether network access is allowed from within the container.

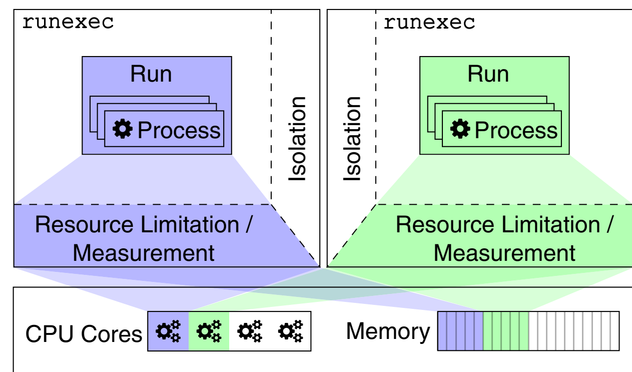

Fig. 6 Resource control and process isolation by `runexec`; a run can consist of many processes; the vertical bars "Isolation" illustrate that each run is executed in isolation and protected to not access other runs; the horizontal bars "Resource Limitation / Measurement" illustrate that the resources are controlled by cgroups and a run can access only the explicitly assigned resources

If `runexec` is used as a stand-alone tool, the inputs are passed to `runexec` as command-line parameters. Alternatively, `runexec` can be used as a Python module for a more convenient integration into other Python-based benchmarking frameworks.

An example command line for executing a tool on all 16 (virtual) cores of the first CPU of a dual-CPU system, with a memory limit of 16 GB on the first memory bank, and a time limit of 100 s is:

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

The output of `runexec` then looks as follows (log on `stderr`, result on `stdout`):

```
10:35:35 - INFO - Starting command <TOOL_CMD>
10:35:35 - INFO - Writing output to output.log
exitcode=0
returnvalue=0
walltime=1.51596093178s
cputime=2.514290687s
memory=130310144
```

In this case, the run took 1.5 s of wall time, and the tool used 2.5 s of CPU time and 130 MB of RAM before returning successfully (return value 0). The same could be achieved by importing `runexec` as a module from within a Python

[23] https://github.com/sosy-lab/benchexec/blob/master/doc/INSTALL.md

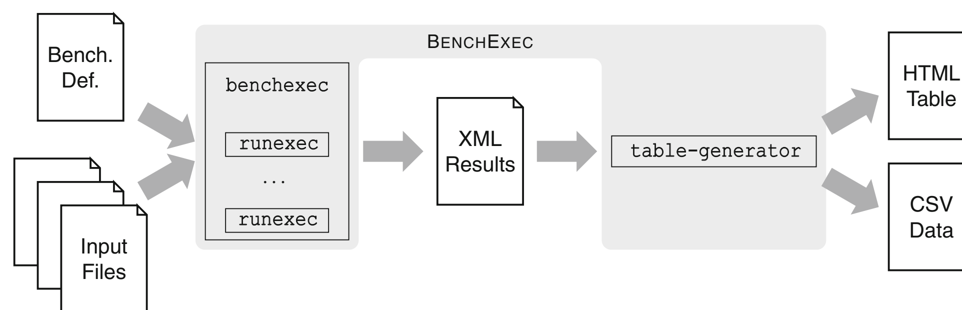

Fig. 7 Benchmarking with `benchexec`; `benchexec` supports the user by automating the execution of experiments with many runs; besides the benchmarked tool, it expects the benchmark definition and a set of input files; `benchexec` executes a series of runs, which are handled (each separately) by `runexec` (cf. Fig. 6); the result is an XML file that contains the raw result data; convenient postprocessing is possible using `table-generator`, which creates customized CSV files and data visualizations (tables and plots) based on HTML

program with a code snippet as in Listing 1 of Appendix A, which returns a dictionary that holds the same information as the key-value pairs printed to `stdout` in the example above. The precise meaning of each of these values is explained in the BenchExec documentation.[24]

6.3 Benchmarking a set of runs

Benchmarking typically consists of processing runs on hundreds or thousands of input files, and there may be several different tools or several configurations of the same tool that should be executed on the same input files. BenchExec provides two programs that allow to perform such large experiments and analyze the results as easily as possible. An overview over the process of using these programs can be seen in Fig. 7.

`benchexec` is a program that executes a set of runs. It receives as input a benchmark definition, which consists of the following components:

- a set of input files,
- the name of the tool to use,
- command-line arguments for the tool (e.g., to specify a tool configuration),
- the limits for CPU time, memory, and number of CPU cores, and
- the number of runs that should be executed in parallel.

This benchmark definition is given in XML format; an example is available in the tool documentation[25] and in Listing 2 of Appendix B. Additionally, a tool-info module (a tool-specific Python module) needs to be written that contains functions for creating a command-line string for a run (including input file and user-defined command-line arguments) and for determining the result from the exit code and the output of the tool. Such a tool-info module typically has under 50 lines of Python code, and needs to be written only once per tool. Experience shows that this tool-info module can be written and integrated successfully into BenchExec also by developers that previously were not familiar with BenchExec.[26]

We are often also interested in classifying the results into expected and incorrect answers. BenchExec supports this for SMT solvers that are compliant to the SMT-LIB standard [2], and for the domain of automatic software verification, where it gets as input a property to be verified in the format used by SV-COMP [5][27]. Classification of results for further domains can be added with a few lines of code.

`benchexec` and its benchmark-definition format also support specifying different configuration options for subsets of the input files, as well as several different tool configurations at once, each of which will be benchmarked against all input files (cf. Listing 2 of Appendix B).

The program `benchexec` starts with trying to find a suitable allocation of the available resources (CPU cores and memory) to the number of parallel runs. For this, it first checks whether there are enough CPU cores and memory in the system to satisfy the core and memory requirements for all parallel runs. Then it assigns cores to each parallel run such that (if possible) a single run is not spread across different CPUs, and different runs use different CPUs or at least different physical cores. Measurement problems due to

[24] https://github.com/sosy-lab/benchexec/blob/master/doc/run-results.md

[25] https://github.com/sosy-lab/benchexec/blob/master/doc/benchexec.md

[26] SV-COMP'16 for the first time required all participating teams to contribute such a module for their tool to BenchExec [6], leading to 21 new tools being integrated into BenchExec.

[27] Tools that do not support this format can also be benchmarked. In this case, the property is not passed to the tool, but used only internally by BenchExec to determine the expected result.

NUMA are avoided by letting each run use only memory that is directly connected to the CPU(s) on which the run is scheduled. Thus, benchexec automatically guarantees the best allocation of the available resources that minimizes the nondeterministic performance influences that were shown in Sect. 4 as far as possible for the given number of parallel runs.

Afterward, benchexec uses runexec to execute the benchmarked tool on each input file with the appropriate command line, resource limits, etc. It also interprets the output of the tool according to the tool-info module (if applicable, it also determines whether the result was correct). The result of benchexec is a data set (in XML format) that contains all information from the runs: tool result, exit code, and measurement values. The output of the tool for each run is available in separate files. Additional information such as current date and time, the host and its system information (CPU and RAM), values of environment variables, and the effective resource limits and hardware allocation are also recorded.

table-generator is a program that produces tables from the results of one or more executions of benchexec. If results of several executions are given to table-generator, they are combined and presented in the table in column groups, allowing to easily compare the results, for example, across different configurations or revisions of a tool, or across different tools. Each line of the generated table contains the results for one input file. There are columns for the tool result and measurement values (such as CPU time, wall time, memory usage, etc.). These tables are written in two formats. A CSV-based format allows further processing of the raw data, such as with gnuplot or R for producing plots and statistical evaluations, a spreadsheet program, or LaTeX packages for reading CSV files in order to present results in a paper as described in Sect. 7. The second format is HTML, which allows the user to view the tables conveniently with nothing more than a web browser. The HTML table additionally provides access to the text output of the tool for each run and contains aggregate statistics and further relevant information such as tool versions, resource limits, etc. The presentation of the measurement results in the table follows the recommendations from Sect. 7.5, including the scientifically valid rounding and number alignment. Furthermore, the HTML table is interactive and supports filtering of columns and rows. HTML tables produced by table-generator even allow generating scatter and quantile plots on the fly for selected columns upon user request (for an explanation of such plots, cf. Sects. 7.6 and 7.7, respectively). Examples of such tables can be found on the supplementary web page.[28]

[28] https://www.sosy-lab.org/research/benchmarking#tables

If a tool outputs further interesting data (e.g., for a verifier, this could be time statistics for individual parts of the analysis, number of created abstract states, or SMT queries), those data can also be added to the generated tables if a function is added to the tool-specific Python module which extracts such data values from the output of the tool. All features of the table (such as generating plots) are immediately available for the columns with such data values as well.

6.4 Comparison with requirements for reliable benchmarking

BenchExec fulfills all requirements for reliable benchmarking from Sect. 2. By using the kernel features for resource management with cgroups that are described in Sect. 5.1, BenchExec ensures accurate measuring and limiting of resources for sets of processes (Sect. 2.1), as well as reliable termination of processes (Sect. 2.2). The algorithms for resource assignment that are implemented in benchexec follow the rules of Sects. 2.3 and 2.4, and assign resources to runs such that mutual influences are minimized as far as possible on the given machine, while warning the user if a meaningful resource allocation cannot be done (e.g., if attempting to execute more runs than physical cores are available, or scheduling multiple runs on a CPU with active Turbo Boost). However, note that our experiments from Sect. 4 show that performance influences from certain hardware characteristics cannot be fully avoided even with the best possible resource allocation if runs are executed in parallel. Furthermore, because runexec alone handles only single runs, it cannot enforce a proper resource allocation across multiple runs, and users need to use either benchexec, an own benchmarking framework, or manual interaction for proper resource allocation across multiple runs. For swapping (cf. Sect. 2.5), BenchExec does what is possible without root privileges: it enforces that swap memory is also limited and measured if present, monitors swap usage, and warns the user if swapping occurs during benchmarking.

Isolation of runs as described in Sect. 2.6 is implemented in BenchExec by executing each run in a container with fresh namespaces. The PID namespace prevents the tool from sending signals to other processes, and network and IPC namespaces prevent communication with other processes and systems. The mount namespace allows customizing the file-system layout. This is used to provide a separate /tmp directory for each run, as well as for transparently redirecting all write accesses such that they do not have any effect outside the container. If necessary, the user can choose to weaken the isolation by allowing network access or write access to some directories. Of course, it is then up to the user to decide whether the incomplete isolation of runs will hinder reliable benchmarking.

Additionally, note that, while BenchExec cannot ensure that users always present benchmark results correctly (e.g., in publications), BenchExec follows the recommendations for valid presentation of results that we describe in Sect. 7, i.e., it always uses SI units (cf. Sect. 7.2) and rounds values to a fixed number of significant digits instead of decimal places (cf. Sect. 7.3).

6.5 Discussion

We would like to discuss a few of the design decisions and goals of BenchExec.

BenchExec aims at not impacting the external validity of experiments by avoiding the use of an overly artificial environment (such as a virtual machine) or influencing the benchmarked tool in any way (except for the specified resource limits and the isolation). For example, while BenchExec allocates resources such as CPU cores for runs to avoid influences between parallel runs, it does not interfere with how the assigned resources are used *inside* each run. If tools with multiple threads or processes are benchmarked, an appropriate resource allocation between these threads and processes within the limits of the run remains the responsibility of the user. The use of namespaces by BenchExec allows us to execute the tool as if it was the only program on a machine with no network connection. Both cgroups and namespaces are present and active on a standard Linux machine anyway, and their use does not add significant overhead.

We designed BenchExec with extensibility and flexibility in mind. Support for new tools and result classifications can be provided by adding a few lines of Python code. The program `runexec`, which controls and measures resources of the actual tool execution, can be used separately as a stand-alone tool or a Python module, for example within other benchmarking frameworks. Result data are present as CSV tables, which allows processing with standard software.[29]

We chose not to base BenchExec on an existing container framework such as LXC or Docker because, while these provide resource limitation and isolation, they typically do not focus on benchmarking, and a fine-grained controlling of resource allocation as well as measuring resource consumption may be difficult or impossible. Furthermore, requiring a container framework to be installed would significantly limit the amount of machines on which BenchExec can be used, for example, because on many machines (especially in clusters for high-performance computing) the Linux kernel is too old, or such an installation is not possible due to administrative restrictions. Using cgroups and namespaces directly minimizes the necessary version requirements, the installation effort, and the necessary access rights, and makes it easy to use a fallback implementation without containers but still with reliable resource measurements due to cgroups if necessary.[30] For example, using Docker would require to give root-level access for the benchmarking machine to all users who want to execute benchmarks.[31] All features of Docker that are necessary for reliable benchmarking (such as process isolation) are available in BenchExec as well and are implemented in a way that after a one-time setup, they only need a few specific privileges that can be granted individually and without a security risk.

We use XML as input and output format because it is a structured format that is readable and writable by both humans and tools, and it is self-documenting. Since XML supports comments, users can also use comments in the benchmark-specification and table-definition files to document their setup. We can store customized result data as well as additional meta data in the result file. This allows documenting information about the benchmarking environment, which is important in scientific work because it increases the replicability and trust of the results.

Python was chosen as programming language because it is expected to be available on practically every Linux machine, and it makes it easy to write the tool-specific module even for people that do not have much experience in programming.

For the future, several research directions and extensions of BenchExec are possible. If necessary, one could work on lifting some of the stated restrictions, and, for example, implement techniques for reliable benchmarking of tools with heavy I/O or use of GPU resources. BenchExec relies on the Linux kernel for the precision of its measurements. To rule out remaining potential nondeterministic effects such as the layout of memory allocations, some users might want to increase the precision by running the same experiment repeatedly. This needs to be done manually today, but built-in support could be added, of course. Furthermore, we are interested in measuring the energy consumption of benchmarks.

6.6 Encouraging replicable experiments

Reliable benchmarking results obtained by accurate and precise measurements are necessary but not sufficient for replicable experiments. The experimental setup also needs to be documented precisely enough that the experiment can be performed again by different people (cf. Sect. 7.1).

[29] For example, BenchExec is used to automatically check for regressions in the integration test-suite of CPAchecker.

[30] We successfully use BenchExec on four different clusters, each under different administrative control and with software as old as SuSE Enterprise 11 and Linux 3.0, and on the machines of the student computer pool of our department.

[31] cf. https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface

While careful documentation is ultimately the responsibility of the experimenter, BenchExec is designed such that it encourages and facilitates documentation. For example, BenchExec accepts the benchmark definition with all relevant properties in a text file (XML-based), which can be explained using comments, and which can be easily archived using a version-control system. This strategy is used, for example, by SV-COMP since 2016[32] [6]. BenchExec is also able to automatically store benchmark results in a Git repository. The result files do not only contain the raw measurement values, but BenchExec also documents important information about the experimental setup, including the hardware characteristics (CPU model and RAM size), the operating-system version, values of environment variables, the version number of the benchmarked tool (if available), the resource limits, and the full command line of the executed run. Taken together, these features conveniently allow to document everything related to a series of experiments, for example in a Git repository, which could be shared with collaborators, or even reviewers or the general public.

7 Presentation of results

After benchmarking results have been obtained, it is typically necessary to present them in a way that is both scientifically valid as well as comprehensible for humans. In this section, we first remind the reader of basic, but important formal aspects, which need to be respected when presenting measurement results. Following this, we discuss different methods that are well-suited for comparing two or more tools.

Note that all visualizations of benchmark results in this paper (the plots from Sect. 4 as well as those in this section) have been automatically generated from BenchExec result files using the LaTeX package `pgfplots`.[33] Documentation on how to achieve this is available in the BenchExec repository.[34]

7.1 Required information

In order to make experiments replicable, some information needs to be given about the benchmarking setup. While it is in general infeasible to provide a complete specification of the benchmarking machine (e.g., including versions of every package on the system), [35] the information that is expected to have the most significant potential impact on benchmarking results needs to be given. We typically consider this to include

- CPU model and size of RAM,
- specified resource limits,
- name and version of OS,
- version of important software packages, such as the kernel or runtime environments like the Java VM,
- version and configuration of the benchmarked tool(s), and
- version of the benchmark set (input files).

For replicability it is further a requirement that the benchmarked tool(s) and the input files are made available, e.g., on a supplementary web page, as additional material in the digital library of the publisher,[36] or in designated online repositories.[37] We recommend to also make the raw benchmarking results and used data-evaluation tools and scripts available in order to make not only the experiment but also the data analysis replicable.

7.2 Units

In general, it is recommended to use SI units and prefixes wherever possible. In particular, special care needs to be taken for values measured in bytes, such as memory or file size. The prefixes *kilo* (k), *mega* (M), *giga* (G), etc. are defined as decimal prefixes, i.e., they always represent a factor of 1000 compared to the next smaller prefix. For example, one megabyte (MB) is exactly 1 000 000 bytes. It is incorrect to use these prefixes as representation for a factor of 1024. While this mistake would create an error of already 2.4% for the prefix kilo, the error would increase to 4.9% for values with the prefix mega, and to 7.4% for values with the prefix giga, which would typically be regarded as a significant systematic error.

Using the common decimal prefixes kilo, mega, etc. is preferred. In cases where the binary prefixes *kibi*, *mebi*, *gibi*, etc. with factors of 1024 are more common (such as for amounts of memory), it is also possible to use these prefixes, if the correct symbols KiB, MiB, GiB, etc. are used. Note that for the prefix kilo a lowercase letter is used as symbol, while for all other prefixes that represent an integral factor (including kibi) an uppercase letter is used.

[32] Cf. SV-COMP benchmark definitions at https://github.com/sosy-lab/sv-comp

[33] https://www.ctan.org/pkg/pgfplots

[34] https://github.com/sosy-lab/benchexec/tree/master/contrib/plots

[35] Providing a complete ready-to-use VM would achieve this, but this is typically not suited for replicating performance results.

[36] Cf. the instructions of the publisher, for example https://www.acm.org/publications/policies/dlinclusions, http://www.ieee.org/documents/ieee-supplemental-material-overview.zip, and https://www.springer.com/gp/authors-editors/journal-author/journal-author-helpdesk/preparation/1276#c40940

[37] For example www.figshare.com or www.runmycode.org

7.3 Significant digits

When writing about measurement results or presenting an overview thereof in tables, it is important to define the number of significant digits to be used throughout the presentation. In this context, it is of importance whether or not a data item being presented is prone to errors in measurement or not. For example, one can provide *exact* numbers for the number of tasks solved by a tool, or for the number of lines of an input file. Hence, these numbers can be given with an arbitrary, but fixed number of digits. With exact numbers, it is also acceptable to round data items to allow a more compact presentation. For example, the length of an input program measured in lines of code (LOC) could be given rounded to 1000 lines with the unit kLOC.

For inexact numbers and measures, for example obtained by measuring time or memory consumption, this is different. One has to define how precise a measurement is, and show all results with a constant number of *significant* digits according to the precision of the measurement. For example, the maximal error of a measurement might be 1, or 0.1, or 0.01%, which would lead to two, three, and four significant digits, respectively. As a concrete example, consider a time measurement with an assumed precision of 0.01%. If the value 43.2109 s is measured, this needs to be reported with four significant digits as 43.21 s, whereas the value 432.1098 s would be shown as 432.1 s. Large and small numbers (e.g., 43210 and 0.04321 s) need to be padded with (insignificant) zero digits. Because from the number 43210 s alone it is not clear whether it has 4 or 5 significant digits, the number of significant digits needs to be explicitly stated for each kind of measurement. This can be avoided by using scientific notation (e.g., 4.321×10^4 s), but this might be harder to interpret for human readers.

Note that the number of significant digits is independent from the used unit, and the reported values always contain exactly the same amount of information no matter in which unit they are.

The naive approach of rounding all numbers to a fixed amount of decimal places (e.g., 4.321 and 0.043 s) should not be used. First, for large numbers, this adds too many digits with irrelevant noise and may lead the reader to wrong conclusions about the precision of the measurement. Second, for small numbers, this leads to a large rounding error that can be orders of magnitudes larger than the measurement error.

7.4 Comparison of summarized results: scores and rankings

It is often necessary to compare the achievements of two or more different tools or tool configurations in a summarized quantitative way. The way this can be done depends on circumstances such as the research area and the goal of the experimental evaluation. In the following, we discuss summarizing results for cases where the result of a tool for an input file can be categorized as either correct, wrong, or inconclusive (as is common in computer-aided verification), i.e., we ignore cases with quantitative tool results (e.g., the quality of an approximation). Inconclusive results are cases where the tool gave up, crashed, or exceeded its resource limits.

In some communities, a wrong result is considered not acceptable, for example in SAT-COMP [1]. In this case, the tools with wrong results are excluded and tools without wrong results can be compared by their number of correct results. A slightly less strict alternative is to rank tools by comparing first the number of wrong results, and in case of ties the number of correct results. This is for example used by SMT-COMP since 2012 [13].

Sometimes it is desired to define a single measure that summarizes the number of correct and wrong results without penalizing tools with wrong results too strongly, for example, because all available tools potentially might produce some wrong results. A scoring scheme can be used to achieve this and calculate a single score for each tool. A typical scoring scheme is as follows: positive points are given for each correct result, zero points are given for each inconclusive result, and negative points are given for each wrong result [3]. The amount of negative points per wrong result is larger than the amount of positive points per correct result in order to penalize wrong results. For example, in SV-COMP'17 one or two points are given for each correct result and −16 or −32 points are given for each wrong result. Note that in this community, some tasks or results (safety proofs) are considered more difficult than others and lead to a higher number of points, and some forms of incorrect results (unsound results) are considered worse than others and lead to a higher penalty.

Because the definition of the scoring scheme heavily influences the ranking and presentation of the evaluation results, a scoring scheme should be used only if it is generally accepted by the community, for example by adoption in a competition in this area. It is important to always also list the raw numbers of correct and incorrect results (possibly further categorized into unsound and incomplete results, for example) in addition to the score.

7.5 Tables

Presenting and comparing numeric data using tables is a straight-forward approach. Authors should respect the suggestions made above regarding the selection of units and stick to a fixed number of significant digits. The latter often results in situations where different cells of the same column have a different number of decimal places, which at first might be counter-intuitive for a reader. If, however, the data in the

Table 1 Example table with different number formats

fixed decimal places (wrong!)	right-aligned	scientific notation	decimal-aligned
123 498.7600	123 500	1.235×10^{5}	123 500
12 349.8760	12 350	1.235×10^{4}	12 350
1 234.9876	1 235	1.235×10^{3}	1 235
123.4988	123.5	1.235×10^{2}	123.5
12.3499	12.35	1.235×10^{1}	12.35
1.2350	1.235	1.235×10^{0}	1.235
0.1235	0.1235	1.235×10^{-1}	.1235
0.0123	0.01235	1.235×10^{-2}	.01235
0.0012	0.001235	1.235×10^{-3}	.001235
0.0001	0.0001235	1.235×10^{-4}	.0001235
0.0010	0.0009876	9.876×10^{-4}	.0009876
0.0099	0.009876	9.876×10^{-3}	.009876
0.0988	0.09876	9.876×10^{-2}	.09876
0.9876	0.9876	9.876×10^{-1}	.9876
9.8761	9.876	9.876×10^{0}	9.876
98.7612	98.76	9.876×10^{1}	98.76
987.6123	987.6	9.876×10^{2}	987.6

columns are aligned to the decimal point, the presentation becomes intuitive at a glance, because the larger a value the more it grows to the left, while the smaller a value the more it shifts to the right. Using scientific notation would also be a possibility, but again, comparing along a row or a column is less intuitive, because all entries would have equal length. Table 1 presents a comparison of the various formats. The first column labeled "fixed decimal places" contains all numbers rounded to the same number of decimal places. As described above, this is incorrect due to rounding errors and measurement noise. The remaining columns all show correct ways of presenting measurement values (with four significant digits), but vary according to their readability and specifically with regard to the possibility of comparing the magnitude of the numbers visually. Right-aligned numbers and scientific notation can make it difficult to distinguish between values such as 123.5 and 12.35, which appear similarly but differ by an order of magnitude. For numbers aligned on the decimal separator (as in the last column), the order of magnitude of each value corresponds to its horizontal position within its cell, which allows the reader to get a quick visual overview of the results. Note that we drop the leading zero for values smaller than 1 such that this effect is increased. The recommended rounding and alignment can be produced in LaTeX tables automatically with packages such as `siunitx`[38] and `pgfplotstable`.[39] The latter even supports creating correctly formatted tables directly from raw CSV files in one step.

Of course, due to space issues, tables can only be used for a small number of results. For larger numbers of results, such as for hundreds or thousands of input files, only summarized data can be shown, and the full set of results should be made available in some other form, e.g., on a supplementary web page or as additional material in a digital library. In addition to showing summarized data, a further possibility is to also show a selection of the most interesting results, such as the n best and worst results[40], and to use plots to visualize a large number of results. The benefit of plots is that for humans this form of presentation usually provides a quick understanding of the data. The downside is that small differences between results are harder or impossible to identify. However, if comparing a set of hundreds or thousands of runs, such detail needs to be sacrificed and abstracted anyway. Moreover, a plot, quite literally, scales better than a table, and is therefore more suitable for comparisons of large data sets.

7.6 Scatter plots

For comparing two columns of related results, for example time results for two different tools for a common set of input files, scatter plots can be used to give a good overview. Scatter plots are drawn onto a two-dimensional coordinate system with the two axes representing the two result columns (typically of the same unit and range). For each input file, there is one data point with the result of the first tool or configuration as its x-value and the result of the other tool or configuration as its y-value. An example scatter plot with CPU-time results for two different tools can be seen in Fig. 8. We highlighted the data point at (16, 1.1), whose meaning is that there exists an input file for which Tool 1 needed 16 s of CPU time and Tool 2 needed 1.1 s.

In a scatter plot, data points on the diagonal show cases where both tools have the same value, data points in the lower-right half of the plot (below the diagonal) show cases where the value of the tool shown on the y-axis is lower, and data points in the upper-left half of the plot (above the diagonal) show cases where the value of the tool shown on the x-axis is lower, and the distance of a data point from the diagonal correlates to the difference between the results. This allows the reader to easily get a quick overview of the results by looking at the distribution of the data points across the two halves. To further aid this, the diagonal and further parallel lines (for example, the functions $y = x + 10$ and $y = x - 10$, or $y = 10x$ and $y = \frac{x}{10}$ on a logarithmic plot) can be added to the plot.

Compared to a table, scatter plots have the disadvantage that it is not possible to see which input file each data point represents. It is also not possible to show results for input files for which one of the results is missing. Data points where at least one component corresponds to a wrong or inconclusive answer also need to be omitted. Otherwise, it could hap-

38 https://www.ctan.org/pkg/siunitx

39 https://www.ctan.org/pkg/pgfplotstable

40 For examples, cf. Tables 4 and 5 in [9]

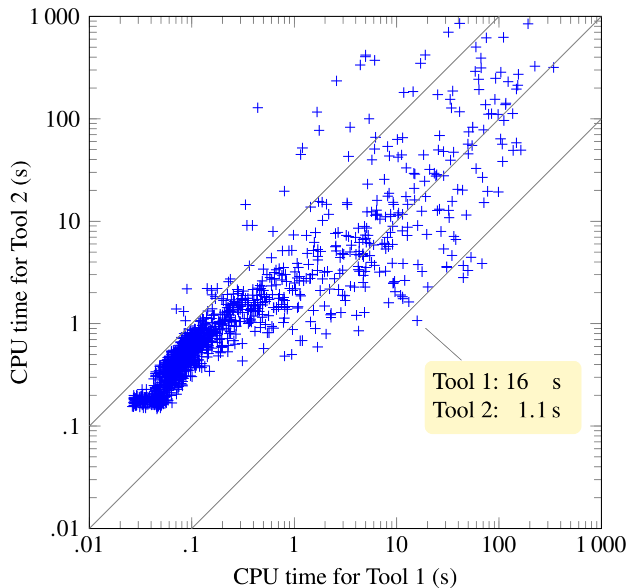

Fig. 8 Example scatter plot, data taken from two tools of category "DeviceDrivers64" of SV-COMP'15 [5]

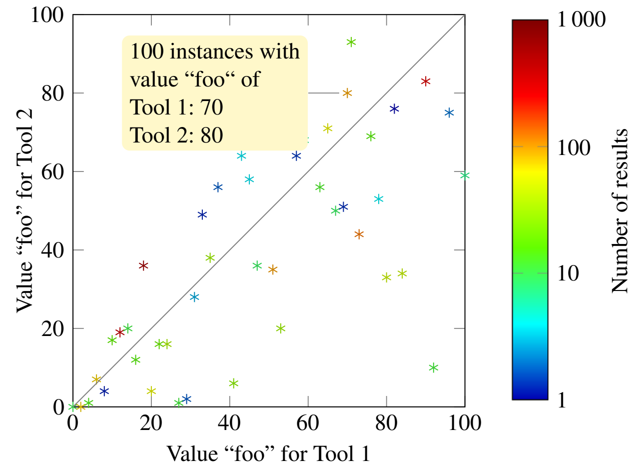

Fig. 9 Example scatter plot with color of data points indicating the number of runs each data point represents

pen that in a scatter plot of time results a tool with many fast but wrong results or fast crashes would visually appear "better" than another tool with slower but correct results, which should be avoided. An exception to this rule are timeout results in scatter plots of time results, which should be shown with a time equal to the time limit and lead (if present in significant numbers) to characteristic clusters at the top or right of the plot. Thus, scatter plots are especially suited for cases where the number of wrong or inconclusive results (except timeouts) is not significant. In other cases it must be clear that the plot shows only a subset of the results and that this can lead to a skewed picture (e.g., if one tool is particularly good for input files for which the other tool crashes, these data points are missing).

Scatter plots support the comparison of only two result columns. More information can sometimes be added by coloring each data point accordingly. However, care should be taken to ensure that crucial information is still available in black-and-white printouts. One use case for a colored scatter plot is to indicate how many results each data point represents, if for example the axes are discrete and there would be cases with many data points at the same coordinates. An example for such a plot can be seen in Fig. 9. For comparing more results, either several scatter plots or different forms of plots should be used.

7.7 Quantile plots

Quantile plots (also known as cactus plots) are plots where the data points represent which quantile (x-value) of the runs need less than the given measure (y-value). Technically, each graph is produced by sorting the data points according to their y-value. This leads to a plot where each graph represents the performance of one tool and the graph is monotonically increasing. Only data points for correct results are shown, and quantile plots are typically used to present time results. Figure 10 shows an example quantile plot with the results of eight tools for 46 input files. The highlighted data point at $(21, 5.4)$ means that Tool 5 produced 21 correct results in at most 5.4 s each and took longer than 5.4 s for its remaining results.

In a quantile plot, the right-most x-value of each graph corresponds to the number of correct results that the respective tool produced. The area under each graph roughly indicates the sum of the run times for all correct results of the tool. Thus, in general, a tool could be considered "better" the further to the right its graph stretches and the lower it is. Furthermore, the slope of a graph may indicate how the tool scales for more difficult tasks. Sometimes, the shape of a graph also gives some further indications. For example, the fact that a tool employs a sequential combination of different strategies and switches strategies after some internal time limit is often recognizable by a characteristic kink in the graph at that time limit. The y-value of the left-most data point of a graph often shows the startup time of the tool. Tools that use a virtual machine, e.g., tools programmed in Java, typically have a slight offset here.

Compared to a scatter plot, quantile plots have the advantage that they can visualize an arbitrary number of result columns, and that data points for all correct results of each tool can be shown (and not only for the intersection of correct results), which means that it also shows the effectiveness of each tool and not only its (partial) efficiency. The disadvantage, however, is that there is no relation between the different graphs and no information regarding how the tools compare on a specific input file. In fact, with a quantile plot

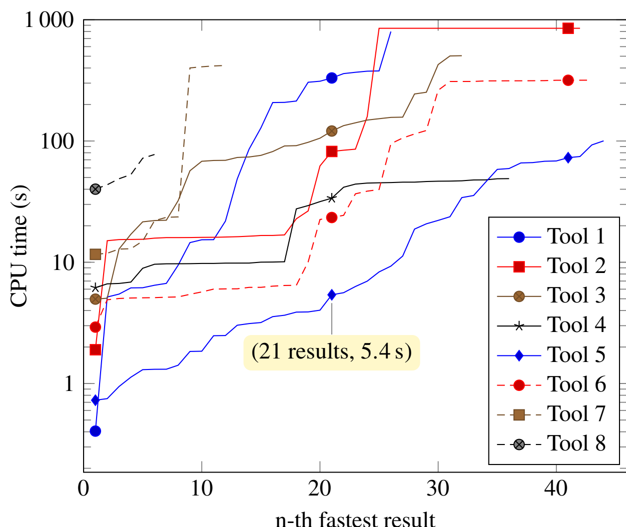

Fig. 10 Example quantile plot, data taken from category "Simple" of SV-COMP'15 [5]

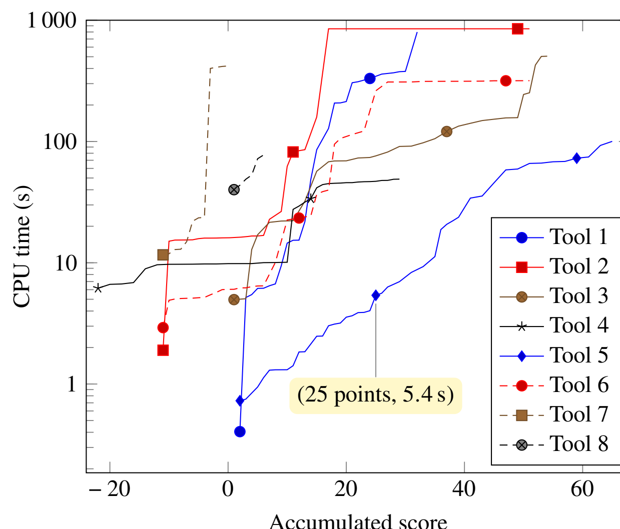

Fig. 11 Example score-based quantile plot, data taken from category "Simple" of SV-COMP'15 [5]

it could well happen that the graphs of two tools are similar or identical, but there is not a single input file for which the two tools produced a similar result.

7.8 Variants of quantile plots

While quantile plots show the number of correct results, they still lack the number of incorrect results. A tool with many correct *and* incorrect results could misleadingly appear "better" than a tool with a lower number of correct results and no incorrect results. This can be avoided by shifting the graph of each tool to the left by the amount of incorrect results. Now the number of incorrect results is visible in the plot as the negative x-offset of the graph, the number of correct results is visible as the x-length of the graph, and their difference is visible as the x-value of the right-most data point. However, this gives the visual indication that an incorrect result and a correct result are somehow of similar importance and cancel each other out, which may not be desired.

This problem can be avoided with a score-based quantile plot as introduced by SV-COMP'13 [4], if a scoring scheme for summarized results is available (cf. Sect. 7.4). In such a plot, a data point (x, y) means that the sum of the scores for all incorrect results and the sum of the scores of all correct results that were faster than y seconds is x. Thus, for example, a correct result with score 2 makes the graph stretch further to the right (and appear "better") than a correct result with the same time but score 1.

In a score-based quantile plot, the accumulated score for all incorrect results is visible as the graph's left-most x-value, the accumulated score for all correct results is visible as the x-length of the graph, and the total score for all results is visible as the x-value of the right-most data point. The general intuition that a graph that is lower and further to the right is "better" still holds. An ideal tool with no wrong results would have its graph start with a positive x-offset.

As example, Fig. 11 shows a score-based quantile plot for the same results as Fig. 10. The highlighted data point corresponds to the same result in both figures, but in Fig. 11 it is drawn at (25, 5.4) because, for the 21 correct results produced by Tool 5 in at most 5.4 s, there were 25 points awarded (and the tool did not produce any wrong answers). This shows how the relative weighting of the results affects the graphs. The influence of incorrect results can be seen, for example, at the graphs of Tool 2, Tool 4, and Tool 6, especially when compared with Tool 3 (cf. the right end of their graphs).

8 Related work

Replicable experiments are an important topic in computer science in general. For example, the ACM has developed a policy on "Result and Artifact Review and Badging" [41] with the goal of standardizing artifact evaluation, and labels articles that support experiment replication with special badges. This work was inspired by effort in this direction in the SIGPLAN and SIGSOFT communities.[42]

Besides the issues that we discussed in this paper, there are more sources of nondeterministic effects that may influence performance measurements in our target domain. For example, the memory layout of the process can affect performance due to differences in cache alignment and cache

[41] https://www.acm.org/publications/policies/artifact-review-badging

[42] http://evaluate.inf.usi.ch/artifacts/aea

hit rates [15], and the memory layout can be influenced by factors such as symbol names [16,21], environment variables and order of objects during linking [24], and nondeterministic memory-allocation strategies of the OS [18].

Benchmarking solutions are also developed in other communities. For example, the MININET HI-FI project [17] provides a framework for experiments in computer networking. It emulates a virtual network on a single machine by using cgroups and namespaces to setup and control a set of virtual hosts and network connections between them.

8.1 Benchmarking strategies

Several strategies have been proposed for dealing with nondeterministic effects that influence measurements by executing runs several times. The execution-time-measurement protocol (EMP) [30] was designed for improving the precision of time measurements by using a sequence of repeated executions and measurements, for example for avoiding the influence of background processes of the operating system. Furthermore, the authors of EMP also identified other sources of measurement errors that the administrator of the benchmarking machine should address. For example, time measurements (even for CPU time) can be imprecise on machines without time synchronization via NTP.

The authors of DATAMILL [25] propose to make benchmarking more reliable by explicitly varying as many hardware and software factors as possible in a controlled manner while benchmarking, e.g., the hardware architecture, CPU model, memory size, link order, etc. To do so, they rely on a diverse set of worker machines, which are rebooted for each benchmark run into a specific OS installation.

In general, these strategies focus on ruling out nondeterministic measurement noise, but do not handle some of the sources of measurement errors identified in this work, such as the CPU-time measurement of tools with several processes, and may thus still produce arbitrarily large measurement errors. On the other hand, BENCHEXEC aims at ruling out such qualitative measurement errors, and relies on the Linux kernel for precise and accurate measurements, but does not increase the precision in case of nondeterministic effects that it cannot control, such as memory-allocation strategies. If such effects are present, users can of course increase the precision by executing benchmarks several times with BENCHEXEC and aggregating the results. In the future, benchmarking with cgroups and namespaces could (and should) be combined with one of the strategies described above to address the respective problem. For example, BENCHEXEC can serve as low-level measurement tool in a distributed framework such as DATAMILL, or EMP could be implemented on top of BENCHEXEC.

8.2 Benchmarking tools

In the verification community, there exist several benchmarking tools that have the same intent as our benchmarking framework. However, as of June 2017, no tool we investigated fulfills all requirements for reliable benchmarking, which are presented in Sect. 2. In the following, we discuss several existing benchmarking tools in their latest versions as of June 2017. Our selection is not exhaustive, because there exist many such tools.

The tool RUNLIM,[43] in version 1.10, allows benchmarking an executable and limits both CPU time and memory. It samples time and memory consumption recursively for a process hierarchy, and thus cannot guarantee accurate measurements and limit enforcement. The tool cannot terminate a process hierarchy reliably, because it only terminates the main process with `kill`. The tool PYRUNLIM,[44] a port of RUNLIM to the Python programming language, has a few more features, such as setting the CPU affinity of a process, and aims at killing process hierarchies more reliably. However, in the latest version 2.15,[45] it does not use cgroups and also takes sample measurements recursively over process hierarchies, which —like all sampling-based methods— is not accurate.

The CProver Benchmarking Toolkit (CPBM),[46] available in version 0.5, ships helpers for verification-task patch management and result evaluation, and also supports benchmarking. However, the limits for CPU time and memory are enforced by `ulimit`,[47] and thus, the benchmarking is not accurate.

Furthermore, none of the tools mentioned so far attempts to isolate independent benchmark runs from each other. This could be done in addition to using one of these tools, e.g., by executing each run under a fresh user account. This would require a substantial amount of additional implementation effort.

The Satisfiability Modulo Theories Execution Service (SMT-Exec)[48] was a solver execution service provided by the SMT-LIB initiative. For enforcing resource limits, SMT-Exec used the tool TREELIMITEDRUN,[49] which uses the system calls `wait` and `setrlimit`, and thus, is prone to the restrictions explained in Sect. 3.

StarExec [29], a web-based service developed at the Universities of Iowa and Miami, is the successor of SMT-Exec. The main goal of StarExec is to facilitate the execution of

[43] http://fmv.jku.at/runlim

[44] http://alviano.net/2014/02/26

[45] GIT revision `b9b2f11` from 2017-05-02 on https://github.com/alviano/python/tree/master/pyrunlim

[46] http://www.cprover.org/software/benchmarks

[47] cf. `verify.sh` in the CPBM package

[48] http://smt-exec.org

[49] http://smtexec.cs.uiowa.edu/TreeLimitedRun.c

logic solvers. The Oracle Grid Engine takes care of queuing and scheduling runs. For measuring CPU time and memory consumption, as well as enforcing resource limits, StarExec delegates to RUNSOLVER[50] [27], available in version 3.3.5, which also is prone to the limitations from Sect. 3.

The Versioning Competition Workflow Compiler (VCWC) [12] is an effort to create a fault-tolerant competition platform that supports competition maintainers in order to minimize their amount of manual work. This project, in the latest development version,[51] defines its own benchmarking container, and uses `sudo` and `schroot`[52] for executing each run in a fresh environment. However, the setup for this needs root access to the machine, and all parallel runs are executed under the same user account, meaning that an executed process can still see and influence processes of parallel runs. Furthermore, VCWC relies on `ulimit` to enforce time limits. If the administrator of the benchmarking machine manually designed and created a cgroup hierarchy that enforces an appropriate partitioning of CPU cores and memory nodes, and defined a memory limit, VCWC can execute runs within these existing cgroups, but it cannot automatically create the appropriate cgroups, as BENCHEXEC does. Furthermore, measurement of CPU time and memory, as well as termination of processes, is not implemented with cgroups, and hence, may fail.

The tool BENCHKIT[53] [22], available in version 1.1, is also used for competitions, where participants submit a virtual-machine (VM) image with their tool and all necessary software. BENCHKIT executes the tool within an instance of this VM and measures the resource usage of the tool and the OS together in the VM. Our framework executes all tools natively on the host system and allows precise measurement of the resource consumption of the tool in isolation, without influence from factors such as the VM's OS. BENCHKIT measures CPU time and memory consumption of the VM using sampling with the performance monitoring tool SYSSTAT[54], which is available in version 11.4.4. BENCHKIT does not ensure that the CPU cores and the memory for a run are assigned such that hyperthreading and NUMA are respected. For each single run with BENCHKIT, i.e., each pair of tool and input file, a new VM has to be booted, which on average takes 40 s to complete [22]. Execution of a tool inside a VM can also be slower than directly on the host machine. Our approach based on cgroups and containers has a similar effect of isolating the individual runs and their resource usage, but comes at practically zero overhead. Our tool implementation was successfully used in SV-COMP'17, which consisted of more than 400 000 runs, plus an uncounted number of runs for testing the setups of the participating tools [7]. An overhead of 40 s per run would have required additional 190 CPU days for the main competition run alone, a prohibitive overhead.

9 Conclusion

The goal of this work is to establish a technological foundation for performance evaluation of tools that is based on modern technology and makes it possible to reliably measure and control resources in a replicable way, in order to obtain scientifically valid experimental results. First, we established reasons why there is a need for such a benchmarking technology in the area of automatic verification. Tool developers, as well as competitions, need reliable performance measurements to evaluate research results. Second, we motivated and discussed several requirements that are indispensable for reliable benchmarking and resource measurement, and identified limitations and restrictions of existing methods. We demonstrated the high risk of invalid measurements if certain technical constraints are not taken care of. Benchmarking problems have been detected in practice, and nobody knows how often they went unnoticed, and how many wrong conclusions were drawn from flawed experiments. Third, we summarized requirements and discussed possibilities for presenting benchmarking results in a scientifically valid and comprehensible way, for example using different kinds of plots.

In order to overcome the existing deficits and establish a scientifically valid method, we presented our lightweight implementation BENCHEXEC, which is built on the concept of Linux cgroups and namespaces. The implementation fulfills the requirements for reliable benchmarking, since it avoids the pitfalls that existing tools are prone to. This is a qualitative improvement over the state of the art, because existing approaches may produce arbitrarily large (systematic and random) measurement errors, e.g., if subprocesses are involved. BENCHEXEC is not just a prototypical implementation. The development of BENCHEXEC was driven by the demand for replicable scientific experiments in our research projects (for the CPACHECKER project, we execute about 3 million tool runs per month in our research lab) and during the repeated organization of SV-COMP. Especially in the experiments of SV-COMP, we learned how difficult it can be to accurately measure resource consumption for a considerable zoo of tools that were developed using different technologies and strategies. BENCHEXEC makes it easy to tame the wildest beast and was successfully used to bench-

[50] http://www.cril.univ-artois.fr/~roussel/runsolver

[51] GIT revision `9d58031` from 2013-09-13 on https://github.com/tkren/vcwc

[52] A utility for executing commands in a chroot environment, cf. http://linux.die.net/man/1/schroot

[53] http://www.cosyverif.org/benchkit.php

[54] http://sebastien.godard.pagesperso-orange.fr

mark 32 tools in SV-COMP'17 [7], with all results approved by the authors of these tools.

Acknowledgements We thank Hubert Garavel, Jiri Slaby, and Aaron Stump for their helpful comments regarding BenchKit, cgroups, and StarExec, respectively, Armin Größlinger for his ideas on what to investigate regarding the performance influence of using multiple CPUs in Sect. 4.8, and all contributors to BenchExec[55].

Appendix A: Machine architectures with hyperthreading and nonuniform memory access

As indication how complex and divergent today's machines can be, we show two exemplary machine architectures, which are shown in Figs. 12 and 13. In the figures, each CPU is represented by a node labeled with "Socket" and the physical package id, each physical CPU core by a node labeled with "Core" and the core id, and each processing unit by a node labeled with "PU" and the processor id. A processing unit is what a process running under Linux sees as a core. Nodes whose label starts with "L" represent the various caches ("L" for level). Contiguous regions of memory are represented by nodes labeled with "NUMANode" and the node id, and each memory region is grouped together with the CPU cores that it belongs to in a green unlabeled node. The numeric ids in the figures are those that the Linux kernel assigns to the respective unit. The numbering scheme is explained in Sect. 3.4. Such figures can be created with the tool `lstopo` from the Portable Hardware Locality (hwloc) project.[56]

Both examples are systems with a NUMA architecture. Figure 12 shows a system with two AMD Opteron 6380 16-core CPUs with a total of 137 GB of RAM. On this CPU, always two virtual cores together form what AMD calls a "module," a physical core that has separate integer-arithmetic units and level-1 data cache for each virtual core, but shared floating-point units, level-1 instruction cache, and level-2 cache. The cores of each CPU are split into two groups of eight virtual cores. The memory of the system is split into four regions, each of which is coupled with one group of cores. This means that, for example, core 0 can access the memory of NUMANode 0 directly and thus fast, whereas accesses to NUMANode 1 would be indirect and thus slower, and accesses to NUMANode 2 and 3 would be even slower as they need to be done via inter-CPU communication channels. Note that on this machine, the two virtual cores of each physical core ("module") got adjacent processor ids and different core ids assigned by the Linux kernel, whereas virtual cores with the same core id on the same CPU actually belong to different physical cores.

Figure 13 shows a system with two Intel Xeon E5-2650 v2 eight-core CPUs with a total of 135 GB of RAM (caches omitted for space reasons). This CPU model has hyperthreading, and thus there are always two virtual cores that share both integer-arithmetic and floating-point units and the caches of one physical core. The memory in this system is also split into two memory regions, one per CPU. Note that the numbering system differs from the other machine: the virtual cores of one physical core have the same core id, which uniquely identifies a physical core on a CPU here. The processor ids for virtual cores, however, are not consecutive but jump between the CPUs.

Note that both presented systems could appear equal at a cursory glance, because they both have the same number of processing units and approximately the same amount of RAM. However, they differ in their architecture and (depending on the workload) could deliver substantially different performance even if running at the same frequency.

55 https://github.com/sosy-lab/benchexec#authors

56 https://www.open-mpi.org/projects/hwloc

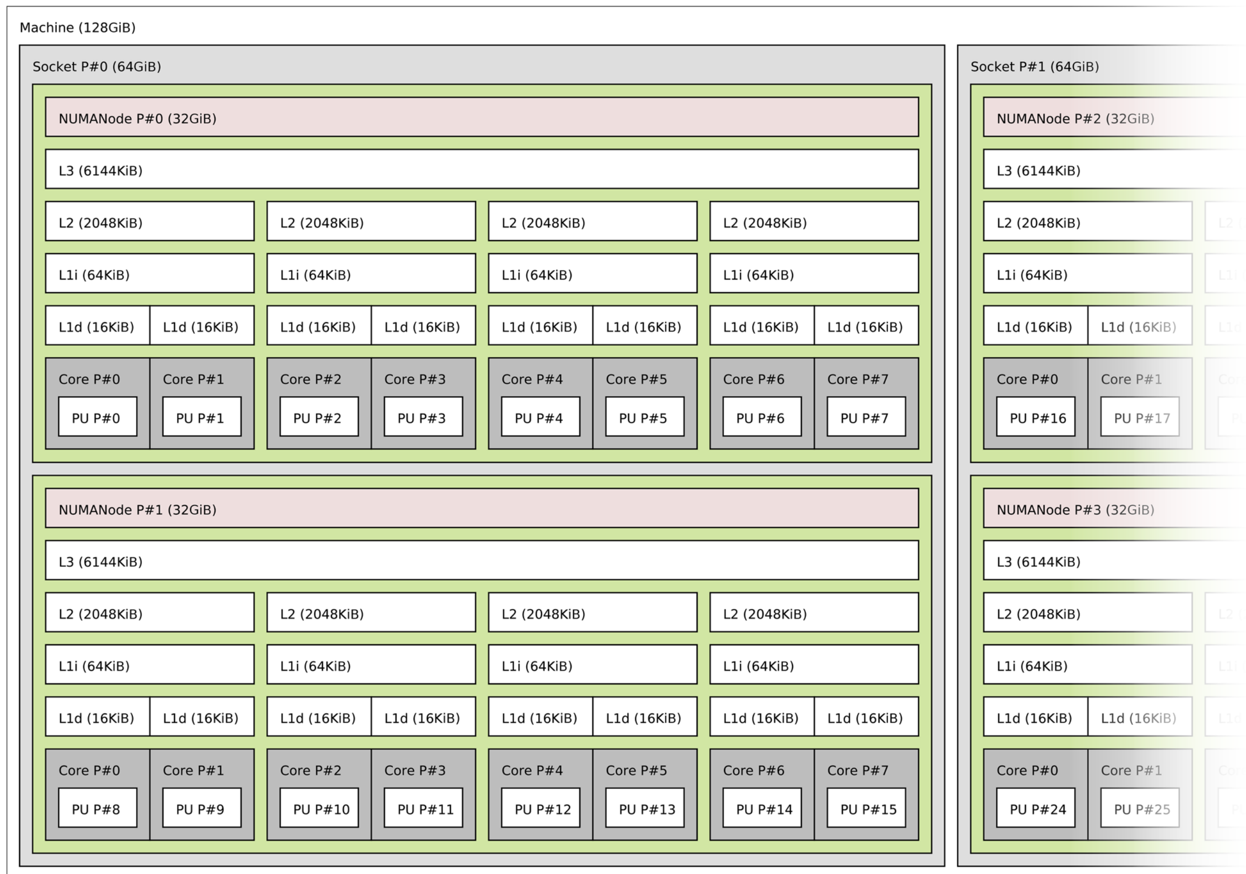

Fig. 12 Example for a machine with a NUMA architecture: 2 AMD Opteron 6380 CPUs, each with two groups of four modules of two cores and 69 GB (64 GiB) of RAM

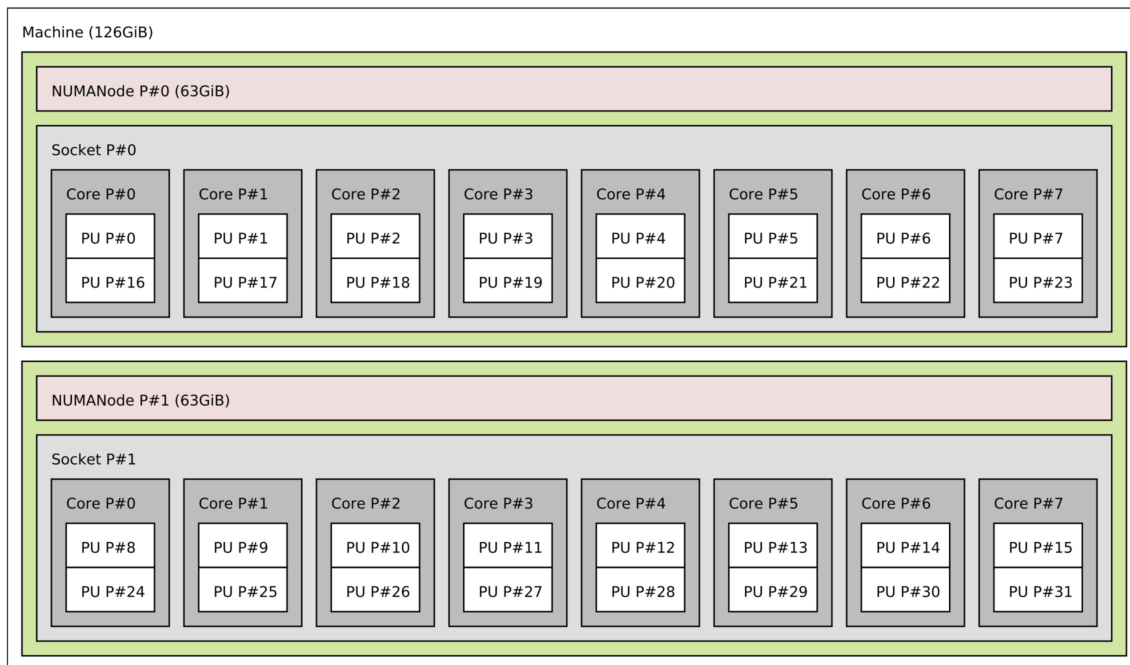

Fig. 13 Example for a machine with a NUMA architecture: 2 Intel Xeon E5-2650 v2 CPUs, each with eight physical cores with hyperthreading and 68 GB (63 GiB) of RAM

Appendix B: Listings

```python
from benchexec.runexecutor import RunExecutor
executor = RunExecutor()
result = executor.execute_run(
            args            = [<TOOL_CMD>],
            output_filename = 'output.log',
            hardtimelimit   = 100,
            memlimit        = 16*1000*1000*1000,
            cores           = list(range(0,8)) + list(range(16,24)),
            memory_nodes    = [0])
```

Listing 1: Example for using module `runexec` from a Python program

```xml
<?xml version="1.0"?>
<!DOCTYPE benchmark PUBLIC
    "+//IDN sosy-lab.org//DTD BenchExec benchmark 1.9//EN"
    "https://www.sosy-lab.org/benchexec/benchmark-1.9.dtd" >

<!-- Example file for benchmark definition, using tool "cpachecker"
     with CPU time limit, memory limit, and 4 CPU cores. -->
<benchmark tool="cpachecker"
           timelimit="900 s" memlimit="8000 MB" cpuCores="4">

  <!-- Define two different configurations to benchmark,
       with different command-line options. -->
  <rundefinition name="predicateAnalysis">
    <option name="-predicateAnalysis"/>
  </rundefinition>

  <rundefinition name="valueAnalysis">
    <option name="-valueAnalysis"/>
  </rundefinition>

  <!-- Global command-line options for all configurations. -->
  <option name="-heap">7000M</option>
  <option name="-noout"/>

  <!-- Define which input files should be used for benchmarking
       (two groups of files declared in separate files). -->
  <tasks name="ControlFlowInteger">
    <includesfile>programs/ControlFlowInteger.set</includesfile>
  </tasks>

  <tasks name="DeviceDrivers64">
    <includesfile>programs/DeviceDrivers64.set</includesfile>
    <!-- These files need a special command-line option: -->
    <option name="-64"/>
  </tasks>

  <!-- Use an SV-COMP property file as specification
    (file ALL.prp in the same directory as each input file). -->
  <propertyfile>${inputfile_path}/ALL.prp</propertyfile>
</benchmark>
```

Listing 2: Example for an XML file as input for program `benchexec`,
defining a benchmark of two configurations of a tool on two sets of files

References

1. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT competition 2016: recent developments. In: Proceedings of AAAI Conference on Artificial Intelligence, pp. 5061–5063. AAAI Press (2017)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.5. Technical report, University of Iowa (2015). www.smt-lib.org
3. Beyer, D.: Competition on software verification (SV-COMP). In: Proceedings of TACAS, LNCS 7214, pp. 504–524. Springer (2012)
4. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proceedings of TACAS, LNCS 7795, pp. 594–609. Springer (2013)
5. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proceedings of TACAS, LNCS 9035, pp. 401–416. Springer (2015)
6. Beyer, D.: Reliable and reproducible competition results with BENCHEXEC and witnesses (Report on SV-COMP 2016). In: Proceedings of TACAS, LNCS 9636, pp. 887–904. Springer (2016)
7. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proceedings of TACAS, LNCS 10206, pp. 331–349. Springer (2017)
8. Beyer, D., Dresler, G., Wendler, P.: Software verification in the Google App-Engine cloud. In: Proceedings of CAV, LNCS 8559, pp. 327–333. Springer (2014)
9. Beyer, D.. Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proceedings of FSE, pp. 389–399. ACM (2013)
10. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proceedings of SPIN, LNCS 9232, pp. 160–178. Springer (2015)
11. Brooks, A., Roper, M., Wood, M., Daly, J., Miller, J.: Replication's role in software engineering. In: Guide to Advanced Empirical Software Engineering, pp. 365–379. Springer (2008)
12. Charwat, G., Ianni, G., Krennwallner, T., Kronegger, M., Pfandler, A., Redl, C., Schwengerer, M., Spendier, L., Wallner, J., Xiao, G.: VCWC: a versioning competition workflow compiler. In: Proceedings of LPNMR, LNCS 8148, pp. 233–238. Springer (2013)
13. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
14. Collberg, C.S., Proebsting, T.A.: Repeatability in computer-systems research. Commun. ACM **59**(3), 62–69 (2016)
15. de Oliveira, A.B., Petkovich, J.-C., Fischmeister, S.: How much does memory layout impact performance? A wide study. In: Proceedings of REPRODUCE (2014)
16. Gu, D., Verbrugge, C., Gagnon, E.: Code layout as a source of noise in JVM performance. Stud. Inform. Univ. **4**(1), 83–99 (2005)
17. Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., McKeown, N.: Reproducible network experiments using container-based emulation. In: Proceedings of CoNEXT, pp. 253–264. ACM (2012)
18. Hocko, M., Kalibera, T.: Reducing performance non-determinism via cache-aware page allocation strategies. In: Proceedings of ICPE, pp. 223–234. ACM (2010)
19. JCGM Working Group 2. International vocabulary of metrology—basic and general concepts and associated terms (VIM), 3rd edition. Technical Report JCGM 200:2012, BIPM (2012)
20. Juristo, N., Gómez, O.S.: Replication of software engineering experiments. In: Empirical Software Engineering and Verification, pp. 60–88. Springer (2012)
21. Kalibera, T., Bulej, L., Tuma, P.: Benchmark precision and random initial state. In: Proceedings of SPECTS, pp. 484–490. SCS (2005)
22. Kordon, F., Hulin-Hubard, F.: BENCHKIT, a tool for massive concurrent benchmarking. In: Proceedings of ACSD, pp. 159–165. IEEE (2014)
23. Krishnamurthi, S., Vitek, J.: The real software crisis: repeatability as a core value. Commun. ACM **58**(3), 34–36 (2015)
24. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing wrong data without doing anything obviously wrong! In: Proceedings of ASPLOS, pp. 265–276. ACM (2009)
25. Petkovich, J., de Oliveira, A.B., Zhang, Y., Reidemeister, T., Fischmeister, S.: DATAMILL: a distributed heterogeneous infrastructure for robust experimentation. Softw. Pract. Exp. **46**(10), 1411–1440 (2016)
26. Rizzi, E.F., Elbaum, S., Dwyer, M.B.: On the techniques we create, the tools we build, and their misalignments: a study of KLEE. In: Proceedings of ICSE, pp. 132–143. ACM (2016)
27. Roussel, O.: Controlling a solver execution with the RUNSOLVER tool. JSAT **7**, 139–144 (2011)
28. Singh, B., Srinivasan, V.: Containers: challenges with the memory resource controller and its performance. In: Proceedings of Ottawa Linux Symposium (OLS), pp. 209–222 (2007)
29. Stump, A., Sutcliffe, G., Tinelli, C.: STAREXEC: a cross-community infrastructure for logic solving. In: Proceedings of IJCAR, LNCS 8562, pp. 367–373. Springer (2014)
30. Suh, Y.-K., Snodgrass, R .T., Kececioglu, J .D., Downey, P .J., Maier, R .S., Yi, C.: EMP: execution time measurement protocol for compute-bound programs. Softw. Pract. Exp. **47**(4), 559–597 (2017)
31. Tichy, W.F.: Should computer scientists experiment more? IEEE Comput. **31**(5), 32–40 (1998)
32. Visser, W., Geldenhuys, J., Dwyer, M.B.: GREEN: reducing, reusing and recycling constraints in program analysis. In: Proceedings of FSE, pp. 58:1–58:11. ACM (2012)
33. Vitek, J., Kalibera, T.: Repeatability, reproducibility, and rigor in systems research. In: Proceedings of EMSOFT, pp. 33–38. ACM (2011)