Benchmarking and Resource Measurement
Application to Automatic Verification

Dirk Beyer, Stefan Löwe, and Philipp Wendler

University of Passau, Germany

Abstract. Proper benchmarking and resource measurement is an important topic, because benchmarking is a widely-used method for the comparative evaluation of tools for automatic verification, needed by researchers, tool developers, and users, as well as in competitions. We formulate a set of requirements that are indispensable for reliable benchmarking and accurate resource measurements, and discuss the limitations of commonly-used methods and benchmarking tools. Fulfilling these requirements in a benchmarking framework is complex and can (on Linux) currently only be done by using the cgroups feature of the kernel. We provide BenchExec, a ready-to-use, tool-independent, and free implementation of a benchmarking framework that fulfills all presented requirements, making accurate, reliable, and reproducible benchmarking easy. Our framework has proven useful and able to work with a wide range of different tools by its successful use in the International Competition on Software Verification.

1 Introduction

Performance evaluation is an effective and inexpensive method to conduct experiments, 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 verifiers or solvers, are also evaluated using performance benchmarking, i.e., measuring execution time, memory consumption, or other performance characteristics. Benchmarking is necessary for comparing different tools of the same domain, evaluating and comparing different features or configurations of the same tool, or for finding out how a single tool performs on different inputs. The ability to limit resource usage (e.g., memory consumption) of a tool during benchmarking is also a requirement for reproducible and comparable measurements. Additionally, for the data obtained in the experiment to be meaningful, it is a requirement that the results are accurate, reliable, and reproducible. Thus, a benchmarking infrastructure that guarantees reliable data must be used. Also competitions, like SAT-COMP [1], SMT-COMP [2, 3], and SV-COMP [4], rely on exact measuring of resource consumption, and, in order to guarantee fairness, need to enforce the agreed resource limits accurately. For example, in the International Competition on Software Verification (SV-COMP), all tools are limited to 15 min of CPU time

[1] `https://www.spec.org/` [2] `http://www.tpc.org/` [3] `http://nlrp.ipd.kit.edu/`

and 15 GB of RAM [4]. If either of these limits is exceeded, a possible result produced by the tool has to be disregarded.

We define reproducibility of results as the possibility to obtain the same results again (assuming a deterministic tool) by re-running the benchmarks on a machine with the same hardware and the same software versions. Hence, to obtain accurate, reliable, and reproducible benchmark results, "volatile" effects must be avoided. While it may appear that measuring execution time is 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 task, is not enough as this does not allow to compare the resource usage of multi-threaded tools, and may be inadvertently influenced by I/O operations. Instead, measuring the CPU time is more meaningful but also more difficult, especially when multiple processes are involved. Furthermore, characteristics of the machine architecture such as hyper-threading or non-uniform memory access can *non-deterministically* affect results and need to be taken into account in order to produce reliable results. Obtaining reliable statistics about memory consumption is even harder, as the memory used by a process may increase or decrease at any point in time. Similarly, the limits on memory consumption must not be violated *at any point in time* during the execution of the tool. Again, multiple involved processes add further complications. Another important aspect is the potentially huge heterogeneity between different tools in a comparison. Of course, tools are written in different programming languages, require different libraries, may spawn child processes, write to hard disks, or perform other I/O operations. All of this has to be taken into consideration when designing a benchmarking environment, ideally in a way that does not exclude any tools from being benchmarked.

1.1 Contributions

In this work we present the following contributions in order to allow thorough benchmarking for all the scenarios described above:

1. We identify a set of requirements that need to be fulfilled for accurate, reliable, and reproducible benchmarking (Sect. 2).
2. We show that the methods that are typically used for resource measurements and limitations do not fulfill these requirements and that this can be a problem in practice (Sect. 3).
3. We describe how to implement a benchmarking environment on a Linux system, fulfilling these requirements (Sect. 4).
4. We implemented and provide as open source BenchExec, a set of ready-to-use tools that fulfill these requirements for reliable benchmarking and are already used successfully in practice in SV-COMP (Sect. 5).

1.2 Restrictions

In order to guarantee accurate, reliable, and reproducible benchmarking, we need to introduce a few restrictions. However, we argue that these restrictions do not interfere with the requirements of tools from our target domain of automatic verification and similar domains. We only consider the benchmarking of tools that adhere to the following restrictions: The tool is (1) CPU-bound, i.e., when

compared to CPU usage, input and output operations from and to disks 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) does not perform network communication during the execution; (3) does not spread across several machines during execution, but is limited to a single machine; and (4) does not require user interaction.

These restrictions are acceptable, because (1) reading from disk, apart from the input file, is not expected from a tool in our domain. 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 disk I/O anyway (e.g., using pipes). Not supporting network communication is acceptable, because (2) we expect tools not to use any network communication. While it would be possible for a tool to offload verification work to remote servers, this also would open the door for "cheating" during competitions. In contrast to other ways that are shown in this paper that may allow circumventing limits imposed by the benchmarking framework, using network communication does not occur accidentally. Benchmarking a distributed tool (3) is much more complex and out of scope. However, techniques and ideas from this paper as well as our tool implementation can be used on each individual host as part of a distributed benchmarking framework.

We do not take into account security considerations. We assume the executed tool to be trusted, i.e., it will not maliciously try to interfere with measurements or other running processes. This could be handled by running our benchmarking framework and the tool under different user accounts, but for this the benchmarking framework needs additional rights (usually root access) that may not be available in all environments. We also do not consider the task of providing the necessary execution environment to the tool, i.e., the system administrator of the machines has to ensure that all necessary packages and libraries needed to run a tool are available in the correct versions. Furthermore, we do not consider shared cloud environments [5], but assume that a dedicated machine is used for benchmarking (at least during the benchmarking), without any CPU- or memory-intensive tasks running outside the control of the benchmarking environment, and that enough memory is installed to handle the operating system, the benchmarking environment, and the benchmarked process(es) without swapping. All I/O is assumed to be local, as network shares can have unreliable performance.

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

2 Requirements for Reliable Benchmarking

There exist two major difficulties that we need to consider for benchmarking. The first is that a tool may spawn arbitrary child processes, and a benchmarking framework needs to handle this. This is indeed common for tools. Examples for child processes that might be started by a verifier include preprocessors, such as

CPP, or solvers, like an SMT-backend. 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. Thus, a significant amount of the verification load (and the resource usage) can happen in one or many child processes that run sequentially or in parallel. Even if it is known that a given tool will not 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.[4]

The second problem occurs when the benchmarking framework should assign specific hardware resources to tool runs, especially if such runs are executed in parallel and the resources need to be divided between them. Current machine architectures can be complex and a sub-optimal resource allocation can negatively affect the performance and lead to non-deterministic and thus non-reproducible results. Examples for differing machine architectures can be seen on the supplementary web page.[5]

In the following, we list five specific requirements that take these problems into account and need to be followed for reliable benchmarking. This list should serve as a checklist not only for implementors of benchmarking frameworks, but also for assessing the quality of experimental results.

2.1 Measure and Limit Resources Accurately

Time. The CPU time of a tool must be measured and limited accurately, including the CPU time of all child processes started by the tool.

Memory. For benchmarking, we are interested in the peak resource consumption of a process, i.e., what is the smallest amount of resources with which the tool could successfully be executed. Thus the memory usage of a process should be defined as the peak size of all memory pages that occupy some system resources. This means, for example, that the size of the address space of a process should not be measured and limited, because it may be much larger than the actual memory usage, for example 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, however, may be too low if data are stored on the stack, and the so-called resident set of a process (the memory that is currently kept in RAM) does not include pages that are used but swapped out.

If a tool consists of multiple 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.

---

[4] Our experience from competition organization shows that developers of complex verification systems are not always aware of how their system spawns child processes and how to properly kill them.

[5] `http://www.sosy-lab.org/~dbeyer/benchmarking`

Setting a limit on the 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 reproducible results.

2.2 Kill Processes Reliably

If a resource limit is violated, it is necessary to reliably kill 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 run. For the operating system, a core is a processing unit that allows execution of one thread. This means that if the CPU supports hyper-threading (i.e., the execution of several threads at the same time in the same physical CPU core), each of the virtual cores is treated as a separate core (processing unit) by the operating system; i.e., the operating system does not distinguish between virtual cores and physical cores. However, because two threads on different virtual cores in the same physical CPU core can influence the performance of each other, there should never be two simultaneous tool executions on two virtual cores of one physical core (just like there should never be two simultaneous tool executions sharing one virtual core). To show that this is important, we executed benchmarks using the verifier CPAchecker on a machine with hyper-threading and on purpose forced two parallel executions of the verifier on the same physical core. This increased the used CPU time by 41 %. More details on this benchmark can be found in the appendix.

Another restriction that should be followed is that the cores for one run should not be split across several CPUs if the run does not need more cores than one CPU can provide, because communication between cores on the same CPU is faster than between different CPUs, and cores share certain caches.

2.4 Respect Non-Uniform Memory Access

Systems with several CPUs often have an architecture with non-uniform memory access (NUMA), which also needs to be considered by a benchmarking environment. In a NUMA architecture, a single processor or a group of processors 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 processor has to access remote memory, this leads to a performance degradation depending on the load of the inter-CPU connection and the other CPU. Hence, a single run of a tool should be bound to memory that is local to its assigned CPU cores, in order to avoid non-deterministic delays stemming from remote memory access. To show that this is important, we executed benchmarks using the verifier CPAchecker on a machine with two CPUs and NUMA and on purpose assigned the cores of

one CPU and the memory attached to the other CPU to each run of the tool, such that all memory accesses were indirect. This increased the used CPU time by 11 %. More details on this benchmark can be found in the appendix.

2.5 Avoid Swapping

Swapping out memory must be avoided during benchmarking, because it may degrade performance in a non-deterministic 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 operating system can decide to start swapping even when 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 modern operating systems. 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.

3 Limitations of Commonly Used Methods

The tools and methods that are typically used on a Linux system for measuring resource consumption and for enforcing resource limits of processes have several problems that make them unsuitable for accurate benchmarking, especially if multiple processes are involved. Any benchmarking environment needs to be aware of these limitations and avoid using naive methods for resource measurements.

3.1 Measuring Resources May Fail

Time. Measuring the wall time is of course easy with high accuracy using the standard tools and methods that a Linux system and most programming languages provide.

Measurement of CPU time that is based on the CPU times of single processes, for example using a variant of the system call `wait` (which returns the CPU time after the given process terminated), does not accurately include the CPU time of child processes that were spawned by the parent 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 kill the solver process, but usually would not bother to wait for its termination. A tool

that runs different analyses in parallel in child processes would also typically terminate as soon as the first analysis returns a valid result, without waiting for the other analyses' termination.[6] In these cases, a large share of the total CPU time is spent by child processes but not included in the measurement.

Memory. The commonly-used Linux 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 using 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 inaccurate 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 sum up the memory usage of all processes. Thus, without keeping track of every memory page of all processes, manually filtering out pages that do not occupy resources because of lazy allocation, and counting each remaining page exactly once, the measured memory usage may be wrong.

3.2 Enforcing Limits May Fail

For setting resource limits, many 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 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 above. Limiting the resident set size (RSS) is no longer supported.[7]

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 implementation of the tool how such a situation is handled: it might terminate itself, or it might crash, or it might even continuously re-spawn the killed 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, killing all processes if a limit is exceeded, but this is inaccurate and prone to the same race conditions described above for memory measurement.

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

[7] `http://linux.die.net/man/2/setrlimit`

3.3 Killing Processes May Fail

In order to kill a tool and all its child processes, one could try to (transitively) enumerate all its child processes and kill 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 purely accidentally, for example due to a flawed logic for restarting crashed child processes of a 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 also might happen incidentally if a parent process is terminated before the child process. In this case, an incomplete benchmarking environment could miss terminating child processes.

The *process groups* of the POSIX standard (established with the system call `setpgid`) 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.

4 State-of-the-Art Benchmarking with Cgroups

We listed aspects that are mandatory for reliable benchmarking, and explained the flaws of commonly-used methods. In the following, we present a technology that can (and should) be used to avoid these pitfalls.

Control groups (cgroups) are a feature of the Linux kernel for managing processes and their resource usage, which is available since 2007 [11]. Differently from all other interfaces for these tasks, 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 [8], 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 affects and measures the resource consumption of the processes within each cgroup regarding a specific resource. 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` [9], which provides an API for accessing and manipulating

[8] Actually, independent hierarchies are currently supported. We restrict ourselves to the single-hierarchy case because multiple hierarchies are going to be deprecated.

[9] `http://libcg.sourceforge.net/`

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 is not common among programs and can be prevented using basic file-system permissions. Any other action of the process, whether changing the 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. For more details on cgroups, we refer to the documentation.[10]

The following cgroup controllers are relevant for reliable benchmarking:

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

`cpuset` allows to restrict the processes in each cgroup to a subset of the available CPU cores. On systems with more than one CPU socket and NUMA, it allows to restrict the processes to specific parts of the physical memory.

`freezer` allows to freeze 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 all of them the kill signal, and afterwards unfreezing ("thawing") them again. 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` allows to restrict maximum memory usage of all processes together in each cgroup, and to measure 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 it holds for these processes (for example disk caches), and then kills at least one process. Alternatively, instead of killing processes, the kernel can send an event to a registered process, which the benchmarking environment can use to terminate all processes within the cgroup in case of an out-of-memory event. The kernel counts only actually used pages towards the memory usage, and as the accounting of memory is done per memory page, shared memory is automatically handled correctly (every page the processes use is counted exactly once).

The `memory` controller allows to define 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, both limits need to be set to the same value. 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, when reading the peak memory consumption, the value for physical memory plus swap memory should be used. Sometimes, the current memory consumption of a cgroup is not zero

[10] `https://www.kernel.org/doc/Documentation/cgroups/`

even after all processes of the cgroup have been killed if the kernel decided to still keep some pages of these processes in its disk cache. To avoid influencing the measurements of other runs by this, a cgroup should be used only for a single run and deleted afterwards, with a new run getting a new, fresh cgroup.[11]

The numbering system of the Linux kernel (which is also used by the `cpuset` controller when restricting the available CPU cores) for a system with n physical cores is such that continuous ids from 0 to $n-1$ are assigned to the first virtual core (processing unit) of each physical core of all CPU sockets in the system, and, in case there are physical cores with more than one virtual core, the ids from n to $2n-1$ are assigned to the second virtual core of each physical core, and so on. For example, consider a system with 2 CPU sockets with 8 physical cores each and 2 virtual cores (processing units) per physical core. There are 16 physical cores in the system, so ids 0–15 refer to the first virtual core of each of these physical cores, and ids 16–31 refer to the other virtual cores. The ids belonging to the first CPU are 0–7 and 16–23, the ids 8–15 and 24–31 belong to the second CPU. The ids of a pair of processing units that share the same physical core differ by 16 in this machine, e.g., (virtual) cores 0 and 16 belong to the same physical core and should be used together. This information can be extracted from certain files in the directories `/sys/devices/system/cpu/cpu<id>/topology/` or from `/proc/cpuinfo`.

5 BenchExec: A Framework for Reliable Benchmarking

In the following, we describe our implementation of a cgroups-based benchmarking framework that fulfills the requirements from Sec. 2 by using the techniques from Sec. 4. It is available as open source under the Apache 2.0 License on GitHub [12].

BenchExec is split into two parts. The first is responsible for benchmarking a single run of a given tool, including the reliable limitation and accurate measurement of resources. This part is also designed such that it is easy to use from within other benchmarking frameworks. The second part is responsible for benchmarking a whole set of runs, i.e., running one or more tools on a collection of input files by delegating each run execution to the first part, which is responsible for a single run, and then aggregating the results.

5.1 System Requirements

In order to use our cgroup-based benchmarking framework BenchExec, a few requirements are necessary that may demand for assistance by the administrator of the benchmarking machine. Apart from running a Linux kernel, cgroups including the four controllers listed in the previous section must be enabled and the account for the benchmarking user needs the permissions to manipulate (a part of) the cgroup hierarchy. Any Linux kernel version of the last years is acceptable, though there have been performance improvements for the memory

[11] Or clear the caches with `drop_caches`. [12] `https://github.com/dbeyer/benchexec/`

controller in version 3.3 [13], and cgroups in general are still getting improved, thus, using a recent kernel is recommended. 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.

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 own administrative control. By creating a special cgroup for benchmarking and granting rights only for this cgroup, it is also possible for the administrator to prevent the user from interfering with other processes and to restrict the total amount of resources that the benchmarking 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.

5.2 Benchmarking a Single Run

We define a *run* as a single execution of a tool, with the inputs being
- the full command line, i.e., the path to the executable with all configuration arguments plus the name of the input file, and optionally,
- the limits for CPU time, wall time, and memory, and
- the list of CPU cores and memory banks to use.

Executing a run produces the following outputs:
- the exit code of the main process,
- output written to stdout and stderr by the tool,
- the CPU time, wall time, and peak memory consumption used by the tool.

The program `runexec` is able to execute a run given the necessary inputs, providing the outputs and ensuring (using cgroups) adherence to the specified resource limits, reliable cleanup of processes after execution (i.e., no process survives), and accurate measurement of the resource usage. This program is runnable stand-alone, in which case the inputs are passed as command-line parameters. Alternatively, the program is usable as a module from within other Python scripts for a more convenient integration into 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 would be:

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

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

```
2015-01-20 10:35:35 - INFO - Starting command <TOOL_CMD>
2015-01-20 10:35:35 - INFO - Writing output to output.log
exitcode=0
returnvalue=0
walltime=1.51596093178s
cputime=2.514290687s
memory=130310144
```

[13] `http://lwn.net/Articles/484251/`

In this case, the run took 1.5 s, and the tool used 2.5 s of CPU time and about 130 MB of RAM before returning successfully (exit code 0). The same could be achieved from within a Python program with three lines of code by importing `runexec` as a module as explained in the documentation [14].

5.3 Benchmarking a Collection of Runs

Benchmarking typically consists of processing tool runs on hundreds or thousands of input files, and there may be several different tools or several configurations of the same tool that run on the same input files.
The program `benchexec` allows to execute multiple runs. It receives as input

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

These inputs are given in XML format; an example can be seen in the tool documentation [14]. Additionally, 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 (verification) result from the exit code and any output of the tool. Such a module typically has under 50 lines of Python code, and needs to be written only once per tool. We are often also interested in classifying the result into expected and incorrect answers. BenchExec currently supports this for the domain of automatic software verification, where it gets as input a property to be verified in the format used by SV-COMP [4] [15].

As an extension, the script and the input format also allow to specify different configuration options (e.g., the appropriate machine model for each input file) for subsets of the input files, as well as to specify several different tool configurations at once, each of which will be benchmarked against all input files.

The program `benchexec` first tries to find a suitable allocation of the available resources (CPU cores and memory) to the number of parallel runs. It 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 a run is not spread over different CPU sockets and (if possible) different runs do not use virtual cores that belong to the same physical core. For memory, it ensures that enough memory is available for all runs and that every run uses only memory that is directly connected to the CPU socket(s) on which the run is executed (to avoid performance problems due to NUMA). Thus, `benchexec` automatically guarantees meaningful resource allocations.

Afterwards, `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 and determines whether the result was correct.

[14] `https://github.com/dbeyer/benchexec/blob/master/doc/INDEX.md`

[15] Tools that do not support this specification format can also be benchmarked. In this case the specification is used by the script only to determine the expected result.

The result of the script is a table (in XML format) that contains all information about the executed runs: returned result, exit code, CPU time, wall time, and memory usage. 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), and the effective resource limits are also recorded.

The program `table-generator` allows to produce tables from the results of one or more executions of `benchexec`. If several result sets are given, they are combined and presented one per column group in the table, 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 result for one input file. There are columns for the output of the tool, the CPU time, the wall time, the memory usage etc. These tables are written in two formats. A CSV-based format allows them to be imported in further tools, such as gnuplot or R for producing plots and statistical evaluations, a spreadsheet program, or a paper written in LaTeX by using a package for CSV import. The second format is HTML, which allows the user to view the tables conveniently with nothing more than a browser. Furthermore, the HTML table is interactive and generates scatter and quantile plots for selected columns, allows columns and rows to be filtered, and provides access to the text output of the tool for each individual run. Examples of such tables can be seen on the supplementary webpage.[16]

If a tool outputs more interesting data (e.g., 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.

5.4 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 benchmarks by avoiding to use an overly artificial environment (such as a virtual machine) or influencing the benchmarked process in any way (except for the specified resource limits). Resource limitations and measurements are done using the respective kernel features that are present and active on a standard machine anyway.

We designed BenchExec with extensibility and flexibility in mind. Support for other tools and result classifications can be added with a few lines of Python code. The program `runexec`, which does the actual benchmark execution and resource measurement, can be used separately as a stand-alone tool or a Python module, for example within other benchmarking frameworks. Result data is present as CSV tables, which allows processing with arbitrary software.[17]

We choose not to base BenchExec on a container solution such as LXC or Docker because, while these provide resource limitation and isolation, they

[16] `http://www.sosy-lab.org/~dbeyer/benchmarking#tables`

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

typically do not focus on benchmarking and precisely controlling resource allocation as well as measuring resource consumption may be difficult or impossible. Furthermore, requiring a container solution to be installed would limit the amount of machines on which BenchExec can be used significantly, for example because on many machines (especially in bigger HPC clusters) the Linux kernel is too old, or such an installation is not possible due to administrative restrictions. Using cgroups directly minimizes both the necessary version requirements as well as the installation effort and the necessary access rights.[18]

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. For example, users can use comments in the input file. The script can not only store customized result data, but also additional meta data in the result file. This allows to document information about the benchmarking environment, which is important in scientific work because it increases the reproducibility of the results.

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

6 Related Work

For computer networking, the Mininet Hi-Fi project [7] also advocates reproducible experiments in their community. In order to achieve resource isolation of processes that belong to different virtual hosts, the tool relies on cgroups.

Other works identify more sources of non-deterministic effects that may influence performance results and proposes methods to address them, such as size of environment variables and order of objects during linking [9].

In the verification community, there exist several other benchmarking tools that share the same intent and common features as our benchmarking environment. However, as of April 2015, no tool we know 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 April 2015. This selection can impossibly be exhaustive, as there exist many different such tools.

The tool RunLim [19], in version 1.7, allows to benchmark another executable and limits both CPU time and memory. It does so by sampling 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 [20], 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 2.11,

[18] 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 faculty.

[19] `http://fmv.jku.at/runlim/`

[20] `http://alviano.net/2014/02/26/`

it does not use cgroups and also takes sample measurements recursively over process hierarchies, which —as all sampling-based methods— is not accurate.

The Satisfiability Modulo Theories Execution Service (SMT-Exec) [21] was a solver execution service provided by the SMT-LIB initiative. For enforcing resource limits, SMT-Exec used the tool TreeLimitedRun [22]. It uses the system calls `wait` and `setrlimit`, and thus, is prone to the restrictions argued in Sect. 3.

StarExec [12], 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 logic solvers on given input files, both of which may be supplied by the user. 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 [23] [10], available in version 3.3.5, that also is prone to the limitations (Sect. 3).

The CProver Benchmarking Toolkit (CPBM) [24], available in version 0.5, ships helpers for verification-task patch management and result evaluation, and also allows benchmarking. However, the limits for CPU time and memory [25] are enforced by `ulimit`, and thus, accurate benchmarking is not guaranteed.

The Versioning Competition Workflow Compiler (VCWC) [6] 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 [26], defines its own benchmarking container, also relying 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, the scripts of VCWC can execute runs within these existing cgroups, but they cannot automatically create the appropriate cgroups like BenchExec. Furthermore, measurement of CPU time and memory, as well as termination of processes, is not done with cgroups, and hence, may fail.

The tool BenchKit [8], available in version $\beta 2$, 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 operating system in the VM together. 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 operating system. BenchKit measures CPU time and memory consumption of the VM using sampling with the performance monitoring tool sysstat [27]. BenchKit does not ensure that the CPU cores and the memory for a run are assigned such that hyper-threading and

[21] `http://smt-exec.org`
[22] `http://smtexec.cs.uiowa.edu/TreeLimitedRun.c`
[23] `http://www.cril.univ-artois.fr/~roussel/runsolver/`
[24] `http://www.cprover.org/software/benchmarks/`
[25] c.f. `verify.sh` in the CPBM package
[26] git revision 9d58031 from 2013-09-13, c.f. `https://github.com/tkren/vcwc/`
[27] `http://sebastien.godard.pagesperso-orange.fr/`

NUMA are taken into account. 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 [8]. Execution of a tool inside a VM can also be slower than directly on the host machine. Our approach based on cgroups has a similar effect of isolating the resource usage of individual runs but comes at practically zero overhead. Our tool implementation was successfully used in SV-COMP'15, in which 54 000 runs were executed, consuming a total of 120 CPU days [4]. Using BenchKit in this competition would have imposed an overhead of 25 CPU days for the 54 000 runs. When also counting runs that were executed by the competition organizers during the testing phase, the total increases to 170 000 runs — a prohibitive overhead.

7 Conclusion

The goal of our contribution is to establish a technological foundation for performance evaluation of verification tools that is based on modern technology and makes it possible to measure and control resources in an accurate, reliable, and reproducible way in order to obtain scientifically valid experimental data. First, we established reasons why there is a need for such a benchmarking technology. Developers of verifiers or solvers in the area of automatic verification, as well as competitions, rely on reliable performance measurement to evaluate the research results. Second, we motivated and discussed several requirements that are indispensable for accurate, reliable, and reproducible benchmarking and resource measurement, and also identified limitations and restrictions of commonly used methods. We report, using rather simple experiments on a large set of tool runs, how easy it is to invalidate measurements if certain technical constraints are not taken care of. Such problems have been detected in practice, and nobody knows how often they went undetected, and how many wrong conclusions were drawn from flawed benchmarks. 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. The implementation fulfills all the requirements for accurate, reliable, and reproducible benchmarking, as 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 errors of measurement, e.g., if sub-processes occur.

BenchExec is not just a prototypical implementation. The development of BenchExec was driven by the demand for accurate scientific experiments in our research projects (e.g., CPAchecker; we execute about 2 million tool runs per month on our lab machines) and during the repeated organization of the International Competition on Software Verification (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 benchmark 22 tools in SV-COMP'15 [28], with all results approved by the 77 authors of these tools.

[28] list on `http://sv-comp.sosy-lab.org/2015/participants.php`

Acknowledgement. We thank Hubert Garavel, Jiri Slaby, and Aaron Stump for their helpful comments regarding BenchKit, cgroups, and StarExec, respectively.

Appendix: Impact of Hyper-Threading and NUMA

To show that hyper-threading and non-uniform memory access (NUMA) can have a negative influence on benchmarking if not handled appropriately, we executed benchmarks using the predicate analysis of the verifier CPAchecker [29] in revision 15 307 of the project repository [30]. We used 4011 C programs from SV-COMP'15 [4] (excluding categories not supported by CPAchecker) and a CPU-time limit of 900 s. Tables with the full results and the raw data are available on the supplementary webpage.[31]

Note that the actual performance impact will differ according to the resource-usage characteristics of the benchmarked tool. For example, a tool that uses only very little memory but fully utilizes its CPU core(s) will be influenced more by hyper-threading than by non-local memory, whereas for a tool that relies more on memory accesses it might be the other way round. In particular, the results for CPAchecker that are shown here are not generalizable and show only that there is such an impact. As the size of the impact is not predictable and might be non-deterministic, it is important to rule out these factors for reliable and accurate benchmarking in any case.

Impact of Hyper-Threading. To show the impact of hyper-threading, we executed benchmarks on a machine with a single Intel Core i7-4770 3.4 GHz CPU socket (with four physical cores and hyper-threading) and 33 GB of memory. We executed the verifier twice in parallel and assigned one virtual core and 4.0 GB of memory to each run. In one instance of the benchmark, we assigned each of the two parallel runs a virtual core from separate physical cores. In a second instance of the benchmark, we assigned each of the two parallel runs one virtual core from the same physical core, such that both runs had to share the hardware resources of this one physical core. A scatter plot with the results is shown in Fig. 1. For the 2 472 programs from the benchmark set that

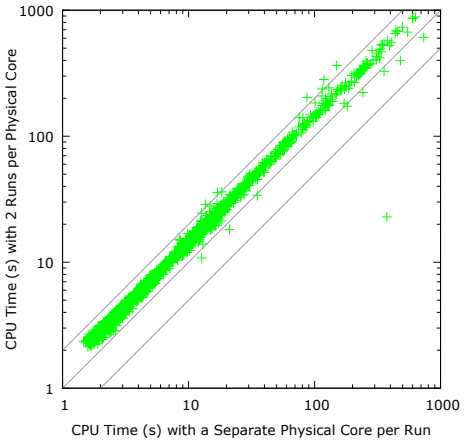

Fig. 1: Scatter plot showing the influence of hyper-threading for 2 472 runs of CPAchecker: data points above the diagonal show a performance decrease due to an inappropriate assignment of CPU cores during benchmarking

[29] http://cpachecker.sosy-lab.org
[30] https://svn.sosy-lab.org/software/cpachecker/trunk
[31] http://www.sosy-lab.org/~dbeyer/benchmarking#benchmarks

CPAchecker could solve on this machine, 13 hours of CPU time were necessary using two separate physical cores and 19 hours of CPU time were necessary using the same physical core, an increase of 41 % caused by the inappropriate core assignment.

Impact of NUMA. To show the impact of non-uniform memory access, we executed benchmarks on a NUMA machine with two Intel Xeon E5-2690 v2 2.6 GHz CPUs with 63 GB of local memory each. We executed the verifier twice in parallel, assigning all cores of one CPU socket and 60 GB of memory to each of the two runs. In one instance of the benchmark, we assigned memory to each run that was local to the CPU the run was executed on. In a second instance of the benchmark, we deliberately forced each of the two runs to use only memory from the *other* CPU socket, such that all memory accesses were indirect. For the 2 483 programs from the benchmark set that CPAchecker could solve on this machine, 19 hours of CPU time were necessary using local memory and 21 hours of CPU time were necessary using remote memory, an increase of 11 % caused by the inappropriate memory assignment. The wall time also increased by 9.5 %.

References

1. A. Balint, A. Belov, M. Heule, and M. Järvisalo. Proc. of SAT competition 2013: Solver and benchmark descriptions. Technical Report B-2013-1, University of Helsinki, 2013.
2. C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2012.
3. C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). *Int. Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.
4. D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.
5. D. Beyer, G. Dresler, and P. Wendler. Software verification in the Google App-Engine cloud. In *Proc. CAV*, LNCS 8559, pages 327–333. Springer, 2014.
6. G. Charwat, G. Ianni, T. Krennwallner, M. Kronegger, A. Pfandler, C. Redl, M. Schwengerer, L. Spendier, J. Wallner, and G. Xiao. VCWC: A versioning competition workflow compiler. In *Proc. LPNMR*, LNCS 8148, pages 233–238. Springer, 2013.
7. N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. CoNEXT*, pages 253–264. ACM, 2012.
8. F. Kordon and F. Hulin-Hubard. BenchKit, a tool for massive concurrent benchmarking. In *Proc. ACSD*, pages 159–165. IEEE, 2014.
9. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. ASPLOS*, pages 265–276. ACM, 2009.
10. O. Roussel. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7:139–144, 2011.
11. B. S. and V. S. Containers: Challenges with the memory resource controller and its performance. In *Proc. Ottawa Linux Symposium (OLS)*, page 209, 2007.
12. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proc. IJCAR*, LNCS 8562, pages 367–373. Springer, 2014.