# *B*ench*K*it, a Tool for Massive Concurrent Benchmarking

Fabrice Kordon
*Sorbonne Universités, UPMC Univ. Paris 06,*
*CNRS, UMR 7606, LIP6, F-75005, Paris, France*
*Fabrice.Kordon@lip6.fr*

Francis Hulin-Hubard
*LSV, CNRS UMR 8643*
*CNRS & ENS Cachan, France*
*francis.hulin-hubard@lsv.ens-cachan.fr*

***Abstract*—Benchmarking numerous programs in a reasonable time requires the use of several (potentially multicore) computers. We experimented such a situation in the context of the MCC (Model Checking Contest @ Petri net) where we had to operate more than 52 000 runs for the 2013 edition.**

**This paper presents *B*ench*K*it, a tool to operate programs on sets of potentially parallel machines and to gather monitoring information like CPU or memory usage. It also samples such data over the execution time. *B*ench*K*it has been elaborated in the context of the MCC and will be used for the 2014 edition.**

***Keywords*-Distributed Computing, Virtual machines, Evaluation of programs**

## I. Introduction

**Context.** These past years, the decreasing cost of computation strength eases intensive software testing or evaluation. More particularly, multi-core machines and/or sets of machines can be solicited for such tasks. The increasing number of concurrent computers systems (multicore, multiprocessor or multi-machines) eases intensive testing or benchmarking.

As a result, several communities have elaborated software competitions aiming at comparing progress of the field. Let us cite first the *Hardware Model Checking Contest* that started in 2007 and now yearly opposes the best hardware verification tools. Other events such as the *Timing Analysis Contest*, the *Verified Software Competition*, the *Satisfiability Modulo Theories Competition* and the *SAT Competitions* have emerged with similar purpose.

In 2011, we started the *Model Checkin Contest @ Petri net* (MCC, http://mcc.lip6.fr) and experimented the difficulty to fairly and appropriately evaluate model checkers from various origins and software architectures.

**Problem.** In that context, benchmarking many programs in a reasonable time requires the use of several (potentially multicore) computers. The execution of processes must be monitored over these machines and centralized for analysis.

Environment such as OAR [1] or Slurm [2] that are now widely used for grid-computing cannot really help for two reasons: *(i)* they are RJMS (Resource and Job Management System) which goal is to handle reservation, access granting in order to use resources, and *(ii)* they require root permission and administration skills to be deployed and operated.

Another tool, `memtime` [3] allows to measure the CPU and memory used by a process but it is restricted to one process and thus cannot report execution when several processes are involved during the execution. This last point is a problem for reproducibility of results by non administrator people [4].

**Contribution.** *B*ench*K*it has been elaborated to solve these problems for the MCC. It allows the deployment of numerous software executions on a set of mono/multicore machines and to consistently report resource use (memory, CPU, I/O, etc.). It is based on virtualization to support the potentially changing execution environments required to operate the programs to be benchmarked. *B*ench*K*it can be operated easily and safely: it does not require `root` permissions to be operated once the virtualization environment installed. It also has a low intrusive footprint and overhead.

So *B*ench*K*it is a flexible solution that proved its efficiency for the MCC and could be operated for tasks requiring the invocation of numerous programs with predefined data for evaluation purpose. It produces CSV files to be exploited using tools such as R [5] or gnuplot [6]. We do not make any assumption on the way programs are used except that they run under Unix-like environments (this includes Cygwin-based execution for Windows).

**Content.** Section II first presents the main issues and constraints raised by monitoring. Section III details the principles of our solution. Section IV presents our experimentation during the model checking contest as well as a discussion on the precision of *B*ench*K*it measures.

## II. Main Issues

Let us first investigate all the issues raised when benchmarking numerous programs like in the MCC.

### A. Benchmark Needs

We consider a benchmark to be a *fair* way to evaluate a software. We do not consider traditional testing where only outputs are compared against a reference but, more precisely, the measure of resource use during the execution. This is why neutrality is important: measure interferences must be as low as possible.

Since establishing a test-bench is a complex task, it is important to provide a configurable environment where the following tasks are facilitated : *(i)* selection of the machine to be used for the benchmark, *(ii)* configuration of these

machines for execution, *(iii)* monitoring of the execution, and *(iv)* the retrieving of monitoring data.

Finally, it is important to enable confinement of the execution (memory, CPU, etc.) for situation where a maximum time/memory is allowed to compute a result.

### B. Typical Execution Support

Today, computers may hold several cores. So a benchmark should target all types of machines and take advantage of available resources. For example, a 4-cores machine could simultaneously host 3 executions if the total available memory allows it in a neutral-way (i.e. swapping or memory controller do not interfere with the execution). It is up to the benchmark designer to configure the resources according to his needs, and the system must enable this.

A crucial point is to preserve *consistency* of benchmarks. As an example, for the MCC, we can decide to use several machines to execute tools on a given benchmark. However, since our goal is to compare tool's execution data, we must ensure that a given input applied on several tools is executed in the same conditions (memory, CPU, etc.). This also stands when using benchmarks in the context of continuous integration: two executions of the same tests for several builds must be operated in similar conditions to emphasize the evolution of performances.

### C. Usability and Security Constraints

Ideally, the system should have the least constraints to be operated by anyone. In terms of operating system, if the environment may be set up with root privileges, it should be operated without such privileges.

As presented in the next section, our solution relies on virtualization. This raises some security issues: the machine supporting the execution should not be exposed to vulnerability. Thus, we must preserve that any execution remains confined in the VM container. A solution suitable for the MCC is to isolate machines, but this may not be a solution for any situation.

### D. Resources Management

This section elaborates our solution to meet the requirements of section II-A in a way that preserves constraints introduced in sections II-B and II-C.

To deal with deployment and execution problems for a standard user, it is necessary to rely on standard solution such as the TORQUE resource manager [7] associated with the MOAD workload manager [8]. These could be used to ensure an exclusive access to the required resources for the benchmark, thus avoiding perturbation in the measures.

Another solution is to rely on the now standardized virtualization mechanisms such as Qemu/KVM that are now integrated in the Linux kernel since February 2007 (2.6.20). Similarly, the `cgroups` system is also a good alternative avoiding an heavy use of virtual machines; this is also standard in the Linux kernel since October 2007 (2.6.24).

Among these solutions, we decided to use virtualization. The main reason is to externalize the installation of the programs to be benchmarked by their developers, thus avoiding to handle at the benchmark level, the dependencies problems for any software to be evaluated. Furthermore, this solution allows the tool developers to be as independent as possible from the constraints of the execution machines.

To facilitate the construction of the disk image for the VM environment, one could use existing facilities such as genvm [9] or Packer [10]. Such solutions for deployment could be integrated in *B*ench*K*it (this is not the case yet).

To cope with security issues when executing a potentially malicious VM, it is easy to disable any access to the network by establishing a dedicated virtual network connected to a dummy interface on the execution computer were only one VM can be executed. For the MCC, *B*ench*K*it needs to log in while there is *a priori* no need for the benchmarked programs to communicate in the other direction. If benchmarked program requires access to the network, other solutions based on forwarding and masquerading could be elaborated. In other situations, one can imagine that code inspection of the tool to be analyzed is performed prior to the benchmark.

## III. *B*ENCH*K*IT

This section details *B*ench*K*it, our generic benchmarking environment that is a response to the needs of section II.

### A. Basic Vocabulary

Let us first introduce the vocabulary used in our framework; it denotes core notions that are related to the execution environment and benchmarking.

**About the execution environment.** We consider four basic notions dealing with the execution environment:

- A virtual node (noted $\mathcal{V}n$) represents a processor core, some memory and disk to run a virtual machine.
- A physical node (noted $\mathcal{P}n$) is a computer hosting one or more $\mathcal{V}n$ (there can be less $\mathcal{V}n$ than the actual number of cores in the physical node).
- A federation of nodes (noted $\mathcal{F}$) is a set of $\mathcal{P}n$, that one may associate to a benchmark.
- A class of nodes (noted $\mathcal{C}$) denotes a set of physical nodes linked by common characteristics. For example, a class "x86_64" could gather all 64-bit X86 processors; similarly, "16GB+" could represent the subset $\mathcal{C} \subseteq \mathcal{F}$ where all $\mathcal{P}n$ with at least 16 GB of memory.

To summarize, $\mathcal{P}n \in \mathcal{C} \subseteq \mathcal{F}$. There are two hierarchical levels since $\mathcal{V}n$ are hosted by $\mathcal{P}n$ (there can be several $\mathcal{V}n$ for a given $\mathcal{P}n$).

Thus, the virtual node is the elementary resource that executes programs. *B*ench*K*it has to dispatch the benchmark execution over $\mathcal{V}n$ of a federation $\mathcal{F}$. Classes are useful to preserve benchmark consistency when deploying the executions.

**About benchmarking.** Four useful notions help to describe the benchmark itself:

- $P$, is a program to be benchmarked,
- $I$, is an input set of data to be processed,
- $\mathcal{E}$, is an examination to be passed by a program, it usually corresponds to a function operated by $P$,
- $\mathcal{R} = \langle P, \mathcal{E}, I \rangle$, is a run corresponding to the execution of a $P$ for an $\mathcal{E}$ with a given $I$. To ensure traceability, runs are associated to a unique identifier *rid*.

To summarize, a benchmark is a set of runs, each one characterizing a situation where the program to be benchmarked has to be confronted.

### B. The Benchmark and its execution

Based on these definitions, a complete benchmark operation is composed of three steps:

1) *preparation*, where the list of runs to be executed is constructed
2) *execution*, that corresponds to the deployment and execution of these runs on the selected machines,
3) *post-processing*, where monitoring information collected during the previous step is analyzed.

*B*ench*K*it is intended for the *execution* step only, that creates a relation between the execution environment and the elements constituting the benchmark. However, the other steps are also facilitated by *B*ench*K*it.

The preparation step is easy : a directory with data must be associated for each input $I$. Then, $\mathcal{E}$ is translated as a way to invoke $P$ with the appropriate link to this directory.

The post-processing phase is also facilitated: produced outputs are standardized in CSV files, their analysis for a given purpose can be extended or easily adapted. Default displays are provided (CPU, memory, I/O) as a facility that can be adapted by users according to their needs.

### C. Functional Architecture

To perform the execution step, we need three types of operations in *B*ench*K*it (see Fig. 1).

**Configuration.** Configuration operations first allow one to set-up the list of machines to be used for the benchmark. It supports exploration functions and automatic detection of the main characteristics of each machine that has been selected. Then, by means of dedicated commands, this configuration can be adapted, for example to use only a subset of the available cores in this machine, or to set-up formulas that, based on the main characteristics of the machines, will dynamically define classes of machines for consistent executions.

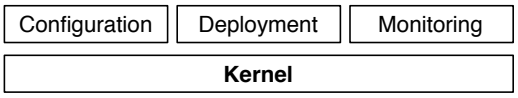


Figure 1. The *B*ench*K*it functional architecture

**Deployment.** These operations allow one to install *B*ench*K*it in the selected $\mathcal{P}n$ (physical nodes) as well as to prepare data to be used on the $\mathcal{P}n$ when the benchmark is running (*e.g.* sending the required disk images, setting up directories to store data, etc.).

**Monitoring.** These operations allow to launch the benchmark on the selected physical machines. This module also offers ways to follow the execution (e.g. how many are already processed) and perform emergency operations such as pausing/resuming/halting the execution if needed.

These three modules rely on a Kernel that operates the executions on all target $\mathcal{P}n$.

### D. Preparation of an Execution Virtual Machine

As explained in section II-D, we decided to rely on virtualization technology. We do not target a given virtualization technology; so far, *B*ench*K*it runs on top of Qemu/KVM. A preliminary version also works on top of VirtualBox.

Thus, for a given $P$ to benchmark, one has to prepare a disk image to be run in a virtual environment. This disk image contains: the (minimal) OS required to execute $P$, directories with data associated to each input $I$, the program $P$, a *B*ench*K*it head.

The *B*ench*K*it head serves as an interface between *B*ench*K*it and $P$. It contains user-defined elements. Its structure is detailed in the next section.

### E. The B*ench*K*it Kernel*

This central part of *B*ench*K*it is executed on the $\mathcal{P}n$ and handles the execution of the programs to be benchmarked in the enclosed $\mathcal{V}n$. There is one instance of this kernel on each $\mathcal{P}n$ in the federation. This kernel locally stores in a FIFO the runs to be executed in the $\mathcal{V}n$. The kernel is composed of two parts:

- the *executor*, located in a $\mathcal{P}n$, that handles executions of runs within a VM with the selected confinement,

**Algorithm 1** The *B*ench*K*it executor in the kernel

**Require:** $v$, the number of available $\mathcal{V}n$
**Require:** $f$, the FIFO containing the runs to be processed
  $nb_{run} \leftarrow 0$
  **repeat**
    **while** ($f$ is not empty) $\wedge$ ($nb_{run} \leq v$) **do**
      $r \leftarrow$ `getEntry`($f$)
      $nb_{run} \leftarrow nb_{run} + 1$
      Start a VM containing *r.P* in background
      Start monitoring function in this VM
      Launch *r.P* with option $r.\mathcal{E}$ and input $r.I$ in the VM
    **end while**
    Wait for the end of a run *rid* (normal or on confinement)
    Retrieve data from *rid*' VM into the database
    Stop the VM associated to *rid*
    $nb_{run} \leftarrow nb_{run} - 1$
  **until** ($f$ is not empty) $\vee$ ($nb_{run} > v$)

**Algorithm 2** Structure of the *B*ench*K*it program head

**Require:** $r.P, r.\mathcal{E}, r.I$
**Ensure:** production of outputs from the invocation of $r.P$
Set up the environment execution ($r.P, r.\mathcal{E}, r.I$, *rid*)
Redirect standard outputs and errors into log files
Go to the directory containing data for $r.I$
Set up a time stamp associated to the program's start
**`User-defined code`** to launch $r.P$ with $r.\mathcal{E}$ options
Setting up a time stamp associated to the program's end

- the B*ench*K*it head*, located in a VM, that launches the executable file in the appropriate directory containing all the data required for the selected run.

**The Executor.** It is basically structured in two loops (see Algorithm 1). The outer one ensures an appropriate termination of all the runs that have been launched. The inner loop handles the runs that remain in the FIFO $f$ associated to the current physical node. We assume that `getEntry` extracts the first entry ($\langle P, \mathcal{E}, I \rangle$) from the FIFO.

**The *B*ench*K*it head.** This is the interface between *B*ench*K*it and the program to be executed. The advantage is to let our environment being independent from the benchmark itself. Thus, the execution of *r.P* in Algorithm 1 is performed thanks to the *B*ench*K*it head.

Algorithm 2 presents the main principles of the *B*ench*K*it head. Basically, it prepares the execution. The user may exploit environment variables to invoke his tool: the examination, the tool name (so that several tools or versions can be located in the same virtual machine, thus simplifying deployment preparation, *e.g.* by participants of the MCC).

### *F. Facts About the current Implementation*

The solution presented in this section could be implemented in various environment, even if we had in mind Unix/virtualization mechanisms for the $\mathcal{P}n$.

If one uses appropriate monitoring solutions, it is easy to monitor any program under any operating system with the appropriate technology embedded on the VM to which *B*ench*K*it may connect to gather its data.

In our experimentation within the model checking contest, we opted for an open source solution based on Linux. So far, the requirements for the $\mathcal{P}n$ are: hardware visualization activate VT (Intel) or SVM (AMD), kernel $\geq$ 2.6.20, KVM activated in kernel and required module: kvm-intel or kvm-amd, Qemu/KVM and the benchmark designer must be in the `kvm` group, Bash $\geq$ 3.0, OpenSSH.

It is recommended to use the local hard drive instead of using a networked file system to avoid latencies.

On the VM executed by the $\mathcal{V}n$ in the $\mathcal{P}n$, we experimented the following setup in 2014: filesystem ext4, Debian wheezy (minbase variant) + network packets, OpenSSH, and SysStat [11] for the monitoring.

A disk image was provided for tool submission (in vmdk for compatibility with other virtualization mechanisms).

## IV. Experimentation

We report here our experience in using *B*ench*K*it for the MCC'2013 and how we intend to use it for the MCC'2014. We also present some evaluation of the low overhead and deviations one can expect in using our environment.

### *A. Using* B*ench*K*it for the Model Checking Contest*

In the 2013 edition of the MCC, we used three machines that correspond to the variety of those one can find in an organization as mentioned in section II-B:

- *cluster1*: a cluster of 23 $\times$ Intel Xeon 2.4 GHz, 6-cores each with 8 Gbytes of memory. Since we had a memory confinement of 4 Gbytes, we could only exploit one core on the 18 nodes we were allowed to use.
- *quahexa-2*: a machine with 4 $\times$ Intel Xeon 2.66 GHz, 6-cores each. Since this machine was equipped with 128 Gbytes of memory, we could use 23 cores (the last one being preserved for the operating system).
- *ebro*: a machine with 4 $\times$ AMD Opteron 2.7 GHz, 16-cores each. Since this machine was equipped with 512 Gbytes of memory, we could use 63 cores (the last one being preserved for the operating system).

Let us now summarize how we operated *B*ench*K*it[1].

**The preparation phase.** Since we had three computers with different characteristics, we maintained the benchmark consistency by operating all examinations for a given model ($I$) and any submitted tools on the same machine to ensure that comparisons were correct.

In that context, we were confronted to the following amount of elements to elaborate our benchmark:

- $P$ : there was 9 tools submission, and, for one of these, 4 variants, thus leading to 12 programs to evaluate.
- $I$ : there was 255 models, coming with sets of formulas to be processed.
- $\mathcal{E}$ : there was 17 examinations for each input (state space generation, 5 variants of LTL and CTL formula, and finally, 6 variants of reachability formulas).

First, a shell script generated $\mathcal{R}$, the list of runs, associating a unique identifier *rid* to each one. There was a total 52 020 runs to be executed within a confinement of 4 Gbytes of memory and 45 minutes of CPU.

Then, we had to split this list into consistent sets to be operated in several machines. In our case, there were three classes in the federation: $\mathcal{C}_1$ = the 18 reserved $\mathcal{P}n$ of *cluster1* (only 1 $\mathcal{V}n$ per node), $\mathcal{C}_2$ = the *quadhexa* $\mathcal{P}n$ (embedding 23 $\mathcal{V}n$), and $\mathcal{C}_3$ = the *ebro* $\mathcal{P}n$ (embedding 63 $\mathcal{V}n$).

**The execution phase.** *B*ench*K*it completely handled this phase. Our main problem was to detect bad runs due to crashes of the virtualization environment. There was about 4.8% of such runs that had to be reprocessed. A post-mortem

[1]This was an earlier version but main principles remain the same, only some technical/implementation aspects have changed.

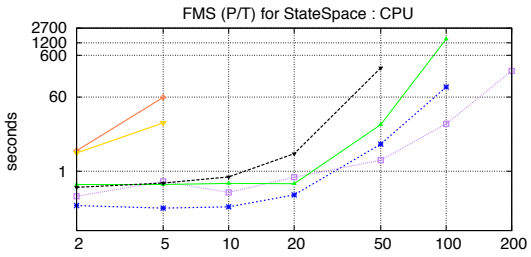


Figure 2. Comparison chart generated from *B*ench*K*it data

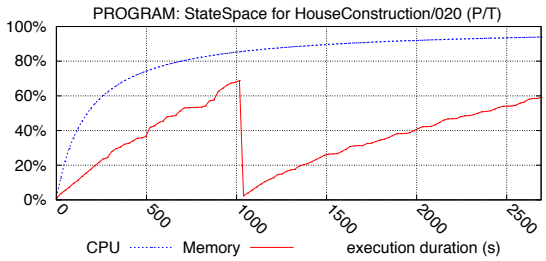


Figure 3. Execution chart generated from *B*ench*K*it data, the name of the measured tool has been removed, confinement is 2700 seconds

analysis revealed that these errors came from some time-out effects when halting a VM that could delay its shutdown, thus preventing the launching of the next one (and leading to a bad run). The use of unique identifiers associated to each run was useful to automatically retrieve and reprocess these defect runs (this could not have been done manually since 4.8% of 52 020 is about 2 500).

The total execution of all runs took 84 days, 6 hours, 2 minutes and 23 seconds of CPU that was operated in about one week. This low performance can be explained as follow:

1) we had numerous configuration troubles with *ebro* that was finally available at the very end of our test (only 1 640 runs could be processed on this machine). Since this computer represented more than 60% of or computation strength, we missed it.
2) we performed a static *a priori* load balancing of the runs. So, we often had to wait for a few $\mathcal{V}n$ while all others were idle,
3) the *B*ench*K*it overhead (see section IV-B), even when the tool declared it was not participating in an examination (this is the only way to know it).
4) Booting/halting time of VMs was not measured.

The new management of the FIFO-run presented in section III-E was performed to solve the load balancing problem. Apart user facilities that have been improved based on our experience during the MCC'2013, this is the major improvement we introduced in *B*ench*K*it that will improve its efficiency on multicore machines.

**The post-processing phase.** Once the process was completed, we got about 2 Gbytes of raw ASCII data: the sampling of memory and CPU consumption all over each execution, a summary of the executions, traces from the executions, trace of the configuration used for each execution.

Shell scripts and dedicated programs exploited these data to produce charts and results that were made available in HTML (http://mcc.lip6.fr/2013) and as a report [12].

Figure 2 shows an example of comparison for various scaling parameters of a model (the legend identifying tools id not provided here). There was 2 093 such charts produced from the raw data we obtained.

Figure 3 shows the variation of CPU and memory usage during one execution (this one failed due to CPU confinement). Sampling of monitored parameters is performed each second. There was 17 284 such charts produced (only the relevant ones where produced, when the corresponding tool invocation stated that outputs would be computed).

### B. Overhead Precision and Stability of Measures

To evaluate the execution duration overhead induced by *B*ench*K*it, the precision and the stability of measures we could observe from one execution to another, we used a `perl` program which duration could be parameterized. This program repeatedly performs memory allocation and deallocation in a deterministic way so that all executions generate the same sequence of actions.

We launched this program natively on a computer representative[2] of those found in any organization. Then, this program was launched on the same machine but within *B*ench*K*it. All measures are averages based on 50 executions.

**Execution Duration Overhead.** To perform a measure, it takes more than just executing the program since we need to: *(i)* boot the VM, *(ii)* launch the monitoring program (SysStat in our case), and *(iii)* retrieve the sampled data. Our experiment revealed that boot time is about 40 seconds and other operations far below 1 second. Thus, the execution duration overhead depends on the execution time of the programs to be benchmarked. The longer it is, the lower the overhead is. If the program lasts 5 minutes, it is about 14% and for a 1 hour length program, it drops to 1.1%.

**Precision of Measures.** To evaluate stability of measures, we operated our test program in both situations, `time` being used to measure execution duration outside *B*ench*K*it.

Figure 4 summarizes the precision of our measures with regards to execution time. We observe that for a 7s calculus, precision is about 1% and drops to 0.1% (that seems to be close to an asymptote) for a 5 minutes length program. However, for shorter programs, the overhead increases and was measured at 7.34% for a 1 second program. This behavior is due to the initial cost of monitoring that is a constant disadvantaging for small programs.

[2]Intel CPU Q6850 @ 3GHz, 4 GB of memory, SATA-II 7 200 rpm disk.

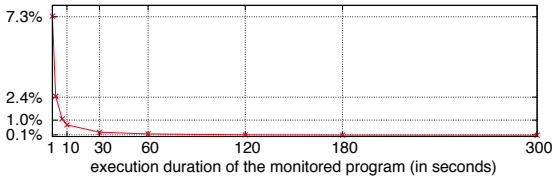


Figure 4. Precision of the CPU measures taken by *B*ench*K*it

So, a constant bias exists and decreases rapidly with execution time. Moreover, it is constant, whatever the program is (at least for the implementation we experimented).

**Stability of Measures.** Figure 5 summarizes our measures of CPU and memory consumption on our program (parameterized to consume 59970 ms of CPU and 227 Mbytes of memory) over 50 executions. CPU is displayed on the left while memory on the right. In both cases, the interval between the measures is low: 0.08% for the CPU and 1.56% for the memory. So, it appears that measures are quite reliable. For other durations, a similar interval was observed (so it appears to be a constant due to the monitoring).

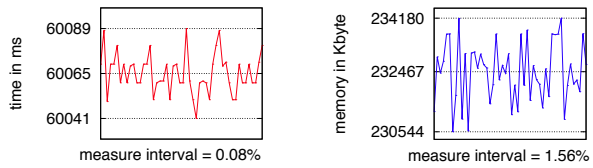


Figure 5. Stability of measures for CPU (left) and memory (right)

### C. Discussion

**Usability aspects.** *B*ench*K*it, elaborated to automate the extraction of monitoring data for the MCC'2013, showed it could handle a large number of benchmarks and let us take benefits from concurrency over several computers or several processors/cores in a single machine.

At this stage, the most challenging problem (out of the *B*ench*K*it scope) is the management and analysis of produced information.

**Reliability of measures.** Even if virtualization and monitoring introduce a bias, it appear that *B*ench*K*it provides quite reliable monitoring information. At least, for our usage in the Model Checking Contest where evaluated programs are more likely to consume a reasonable amount of time and memory, we can consider the bias as being negligible.

**Extra Time Required to Perform Benchmarks.** The extra CPU required to benchmark programs with *B*ench*K*it could be considered as very high. For example it corresponds to 25 extra days for the 84 days of CPU measured for the MCC.

However, two aspects tend to minimize this problem. Firstly, thanks to parallelism, this cost is heavily reduced (the theoretical gain by $N$ is in fact achieved for $N$ machines or cores). On a multicore machine, our kernel most likely ensures that VM are booting while other are executing the program. On a multimachines, several kernel are executed in parallel. So, in fact, the 25 extra days for computation where greatly absorbed by the use of our parallel machines.

Secondly, it is negligible compared to the cost of configuring and installing the participating tools and their (potentially conflicting) dependencies. For the MCC, we avoid having to care with this time consuming task when configuring several machines. Moreover, this task is impossible without root permission.

## V. Conclusion

This paper presented *B*ench*K*it, a generic environment to automatically operate massive parallel benchmarks on various sets of machines that can be organized according to various consistency criteria. We present the principles of such an environment and discuss some potential implementation (Unix based). We showed that, *B*ench*K*it has a small footprint and provides fair and stable measures, except for very short programs (*e.g.* below 1 second where 7% overhead was measured).

*B*ench*K*it is intended to evaluate resource consumption (CPU, memory, I/O, etc.) all over the execution of the program to be benchmarked. It also handles confinement of such programs (max memory or CPU) and stops their execution when needed. There could be various application of such an environment, especially in continuous integration development approaches. We have experimented a prototype version for the model checking contest @ Petri Net in 2013 and the version described in this paper will be used for the MCC'2014. After this event, it will be published under an open source license[3]. At this stage, *B*ench*K*it is more intended for large programs that is the context of the MCC.

Future extensions of *B*ench*K*it should consider the benchmarking of multicore programs (this is technically possible at this stage but requires to extend the configuration interface) as well as the reduction of bias for short programs. Help to the end-user such as VM construction and default data plotting are also being investigated.

## References

[1] "OAR home page," 2013. [Online]. Available: http://oar.imag.fr

[2] "Slurm Workload Manager," 2013. [Online]. Available: http://slurm.schedmd.com

[3] "`memtime`," 2006. [Online]. Available: http://www.it.uu.se/research/group/darts/uppaal/benchmarks/

[4] S. Hostettler, A. Linard, A. Marechal, and M. Risoldi, "Improving the significance of benchmarks for petri nets model checkers," in *Workshops of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS)*, vol. 827. CEUR-WS.org, 2010, pp. 475–489. [Online]. Available: http://ceur-ws.org/Vol-827

[3] See http://benchkit.cosyverif.org

[5] "The R project for statistical computing," 2006. [Online]. Available: http://www.r-project.org
[6] "gnuplot homepage," 2006. [Online]. Available: http://www.gnuplot.info
[7] Adaptive Computing, "TORQUE Resource Manager," 2014. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque
[8] ——, "Moab HPC Suite Basic Edition," 2014. [Online]. Available: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition
[9] "genvm Generate debian virtual machines," 2013. [Online]. Available: http://sourceforge.net/projects/genvm/
[10] Adaptive Computing, "Packer home page," 2014. [Online]. Available: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition
[11] "the SysStat utilities Home Page," 2013. [Online]. Available: http://sebastien.godard.pagesperso-orange.fr
[12] F. Kordon, A. Linard, M. Beccuti, D. Buchs, L. Fronc, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, E. Paviot-Adet, F. Pommereau, C. Rodrigues, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf, "Model Checking Contest @ Petri Nets, Report on the 2013 edition," CoRR, Tech. Rep., September 2013. [Online]. Available: http://arxiv.org/abs/1309.2485

## VI. Appendix A, Online Help for **bklr**

The `bklr` (*B*ench*K*it Launch a Run) launches a VM and executes an examination on a given input for a given machine. This is the main command of *B*ench*K*it. It is used by the scheduler which operates all the tests for a given benchmark.

```
$ bklr -h
-----------------------------------------------------
BenchKit 2
    F. Kordon & F. Hulin-Hubard
-----------------------------------------------------
bklr : BenchKit Launch Run
 $ bklr <disk image> <input name> <examination name>

Options
-c  --config  config file (default : conf/config.txt)
-d  --debug  mode debug
-h  --help this help message
-v  --version display version informations
```

## VII. Appendix B, Execution Trace for **bklr**

Here is a typical execution for the first submitted tool for the Model Checking Contest @ Petri net for 2014. It is performed on an example (FMS-PT-005) for the *StateSpace* examination.

The only constraint for tools is to respect a formatted output to be detected by the post analysis scripts. For this examination, information must start with `STATE_SPACE`.

The `time` command added to the command line shows that, while the execution time of the computation is about 60 seconds, the total execution time (including the overhead for booting the VM) is about 83 seconds. Such an execution overhead is acceptable since it compensates numerous time consuming tasks.

```
$ time ./bklr ../tapaal.vmdk FMS-PT-005 StateSpace
runnning tapaal on FMS-PT-005 (StateSpace)
We got on stdout :
Probing ssh
.................................................
============================================================
 Generated by BenchKit 2-1615
    Executing tool tapaal
    Input is FMS-PT-005, examination is StateSpace
============================================================

--------------------
content from stdout:

START 1398843898
****************************************
*  TAPAAL performing StateSpace search  *
****************************************
STATE_SPACE 2895018 -1 21 5 TECHNIQUES EXPLICIT
STOP 1398843958

--------------------
content from stderr:

--------------------
content from /tmp/BenchKit_head_log_file.1628:

real    1m23.516s
user    0m0.070s
sys     0m0.060s
```