VMPlaceS: A Generic Tool to Investigate and Compare VM Placement Algorithms

Adrien Lebre, Jonathan Pastor, Mario Südholt

ASCOLA Research Group (Mines Nantes, Inria, LINA), Nantes, France
firstname.lastname@inria.fr

Advanced Virtual Machines placement policies are evaluated either using limited scale *in-vivo* experiments or ad hoc simulator techniques. These validation methodologies are unsatisfactory. First they do not model precisely enough real production platforms (size, workload representativeness, etc.). Second, they do not enable the fair comparison of different approaches.

To resolve these issues, we propose VMPlaceS, a dedicated simulation framework to perform in-depth investigations and fair comparisons of VM placement algorithms. Built on top of SimGrid, our framework provides programming support to ease the implementation of placement algorithms and runtime support dedicated to load injection and execution trace analysis. It supports a large set of parameters enabling researchers to design simulations representative of a large space of real-world scenarios. We also report on a comparison using VMPlaceS of three classes of placement algorithms: centralized, hierarchical and fully-distributed ones.

1 Introduction

Most of the popular Cloud Computing (CC) management systems [**?**,**?**,**?**], or IaaS toolkits [**?**], rely on elementary virtual machine (VM) placement policies that prevent them from maximizing the usage of CC resources while guaranteeing VM resource requirements as defined by Service Level Agreements (SLAs). Typically, a batch scheduling approach is used: VMs are statically allocated to physical machines according to user requests. Such static policies are clearly suboptimal, because users often overestimate their needs and the effective resource requirements of a VM may significantly vary during its lifetime [**?**].

An important impediment to the adoption of more advanced strategies such as dynamic consolidation, load balancing and other SLA-enforcing algorithms developed by the academic community [**?**,**?**,**?**,**?**,**?**] is related to the experimental processes used for their validation: most VM placement proposals have been evaluated either using ad hoc simulators or small *in-vivo* (*i.e.*, real-world) experiments. These methods are not accurate and not representative enough to (i) ensure their correctness on real platforms and (ii) perform fair comparisons between them. Analyzing proposals for VM placement on representative testbeds in terms of scalability, reliability and varying workload changes would definitely be the most rigorous way to support their development for CC production infrastructures. However, *in-vivo* experiments, if they can be executed at all, are always expensive and tedious to perform (see [**?**] for a recent reference).

In this article, we propose VMPlaceS, a dedicated simulation framework to perform in-depth investigations of VM placement algorithms and compare them in a fair way. To cope with real conditions such as the increasing scale of modern data centers, as well as the workload dynamicity and elasticity characteristics that are specific to the CC paradigm, VMPlaceS allows users to study large-scale scenarios that involve thousands of VMs, each executing a specific workload that evolves during the simulation. To illustrate the relevance of VMPlaceS, we have implemented and analyzed three well-known approaches: Entropy [**?**], Snooze [**?**], and DVMS [**?**]. We chose these three systems as they represent three classes of placement algorithms: Entropy is an instance of a centralized model, Snooze of a hierarchical one and DVMS of a fully distributed one. Using VMPlaceS, we compare the scalability and reactivity (*i.e.*, the time to solve SLA violations) of the strategies — a contribution of its own. Our results also reveal the importance of the duration of the reconfiguration phase (*i.e.*, the step where VMs are relocated throughout the infrastructure) compared to the computation phase (*i.e.*, the step where the scheduler solves the VMPP). We believe that VMPlaceS will be beneficial to a large number of researchers in the field of CC as it enables them to analyze the main characteristics of a new proposal, allowing *in vivo* experiments to be restricted to placement mechanisms that have the potential to handle CC production infrastructures.

The rest of the article is organized as follow. Sec. **??** gives an overview of the SimGrid framework on which our proposal is built. Sec. **??** introduces VMPlaceS. Entropy, Snooze and DVMS are briefly presented in Sec. **??** and evaluated in Sec. **??**. Secs. **??** and **??** present, respectively, related work and a conclusion.

2 Simgrid, a Generic Toolkit To Build Simulators

SimGrid is a toolkit for the simulation of potentially complex algorithms executed on large-scale distributed systems [**?**]. To perform simulations, users develop a *program*, and define a *platform* and a *deployment* files. The *program* typically leverages SimGrid's MSG API that allows end users to create and execute SimGrid abstractions such as processes, tasks, VMs and network communications. The *platform* file provides the physical description of all resources that are the object of the simulation. The *deployment* file is used to launch the different SimGrid processes of the program on the simulated nodes. Finally, the simulation is orchestrated by the SimGrid engine that internally relies on a constraint solver to assign the CPU/network resources during the entire simulation.

We chose to base VMPlaceS on SimGrid since (i) the latter's relevance in terms of performance and validity has already been demonstrated [**?**] and (ii) because it has been recently extended to integrate VM abstractions and a live migration model [**?**]. In addition to enabling researchers to control VMs in the same manner as in the real world (*e.g.*, create/destroy VMs; start/shutdown, suspend/resume and migrate them), the live migration model provided by SimGrid is the only one that successfully determines correctly the time and the resulting network traffic of a migration by taking into account the competition arising in the presence of resource sharing and the memory refresh rate. These two capabilities were mandatory to build VMPlaceS.

3 VM Placement Simulator

The aim of VMPlaceS is twofold: (i) to relieve researchers of the burden of dealing with VM creations and workload fluctuations when they evaluate new VM placement algorithms and (ii) to offer the possibility to compare them.

Overview. VMPlaceS has been implemented in Java by leveraging the SimGrid MSG API. Although Java has an impact on the efficiency of SimGrid, we believe its use is acceptable because Java offers important benefits to researchers for the implementation of advanced scheduling strategies, notably concerning the ease of implementation of new strategies. As examples, we implemented the Snooze proposal in Java and the DVMS proposal using Scala and Java.

VMPlaceS performs a simulation in three phases, see Fig. **??**: (i) initialization, (ii) injection and (iii) trace analysis. The initialization phase corresponds to the creation of the environment, the VMs and the generation of an event queue. The simulation is performed by at least two SimGrid processes, one executing the *injector*, the generic part of the framework which is in charge of injecting the events during the execution of the simulation, and a second one executing the to-be-simulated *scheduling algorithm*. The latter analyzes the collected traces in order to gather the results of the simulation, notably by means of the generation of figures representing, *e.g.*, resource usage statistics.

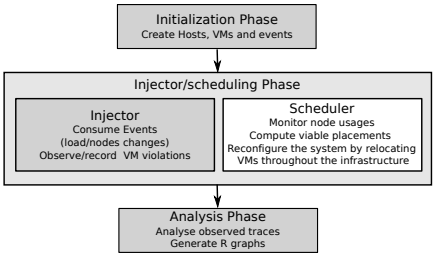


Fig. 1. VMPlaceS's Workflow
Gray parts correspond to the generic code while the white one must be provided by end-users.

Researchers develop their scheduling algorithm using the SimGrid MSG API and a more abstract interface that is provided by VMPlaceS and consists of the classes `XHost`, `XVM` and `SimulatorManager`. The two former classes respectively extend SimGrid's `Host` and `VM` abstractions while the latter controls the interactions between the different components of the simulator. Through these three classes users can inspect, at any time, the current state of the infrastructure (*i.e.*, the load of a host/VM, the number of VMs hosted on the whole infrastructure or on a particular host, check whether a host is overloaded, etc.).

Initialization Phase. VMPlaceS first creates n VMs and assigns them in a round-robin manner to the first p hosts defined in the platform file. The default platform file corresponds to a cluster of $h + s$ hosts, where h corresponds to the number of hosting nodes and s to the number of services nodes. The values n, h and s constitute input parameters of the simulations (specified in a Java property file). These hosts are organized in form of topologies, a cluster topology being the most common one. It is possible, however, to define more complex platforms to simulate, for instance, federated data center scenarios.

Each VM is created based on one of the predefined VM classes. A VM class corresponds to a template specifying the VM attributes and its memory footprint. It is defined in terms of five parameters: the number of cores `nb_cpus`,

the size of the memory `ramsize`, the network bandwidth `net_bw`, the maximum bandwidth available `mig_speed` and the maximum memory update speed `mem_speed` available when the VM is consuming 100% of its CPU resources. Available classes are defined in a text file that is modifyable by users. As pointed out in Section **??**, the memory update speed is a critical parameter that governs the migration time as well as the amount of transferred data. VM classes provide means to simulate arbitrary kinds of workload (*e.g.*, memory-intensive ones)

All VMs start with a CPU consumption of 0 that will evolve during the simulation depending on the injected load as explained below. Once the creation and the assignment of VMs completed, VMPlaceS spawns at least two SimGrid processes, the *injector* and the launcher of the selected scheduler. At its start the *injector* creates an event queue that will be consumed during the second phase of the simulation. Currently, VMPlaceS supports *CPU load change* events (only). The event queue is generated in order to change the load of each VM every t seconds on average. t is a random variable that follows an exponential distribution with rate parameter λ_t while the CPU load of a VM evolves according to a Gaussian distribution defined by a given mean (μ) as well as a given standard deviation (σ). t, μ and σ are provided as input parameters of a simulation. Furthermore, each random process used in VMPlaceS is initialized with a seed that is defined in a configuration file. This way, we can ensure that different simulations are reproducible and may be used to establish fair comparisons.

Finally, we highlight that adding new events can be done by simply defining new event Java classes implementing the `InjectorEvent` interface and by adding the code in charge of generating the corresponding events that are then handled similarly to the *CPU Load* ones.As an example, the next release of VMPlaceS will integrate *node apparition/removal events* that will be used to simulate crashes.

Injector Phase. Once the VMs and the global event queue are ready, the evaluation of the scheduling mechanism can start. First, the injector process iteratively consumes the different events. Changing the load of a VM corresponds to the creation and the assignment of a new SimGrid task in the VM. This new task has a direct impact on the time that will be needed to migrate the VM as it increases or decreases the current CPU load and thus its memory update speed.

Based on the scheduler decisions, VMs will be suspended/resumed or relocated on the available hosts. Users must implement the algorithm in charge of solving the VMPP but also the code in charge of applying reconfiguration plans using methods from the `SimulatorManager` class. This step is essential as the reconfiguration cost is a key element of dynamic placement systems.

It is noteworthy that VMPlaceS invokes the execution of each scheduling solver, as a real implementation would do, to get the effective reconfiguration plan. That is, the computation time that is observed is not simulated but corresponds to the effective one, only the workload inside the VMs and the reconfiguration operations (*i.e.*, suspend/resume and migrate) are simulated in SimGrid.

Trace Analysis. The last step of VMPlaceS consists in analyzing the information that has been collected during the simulation. This analysis is done in two steps. First, VMPlaceS records several metrics related to the platform utilization

using an extended version of SimGrid's TRACE module[1]. This way, visualization tools that have been developed by the SimGrid community, such as PajeNG [**?**], may be used with VMPlaceS. Furthermore, our extension enables the creation of a JSON trace file, which is used to represent resource usage by figures generated using the R statistical environment [**?**].

By default, VMPlaceS records the load of the VMs and hosts, the start and the duration of each violation of VM requirements in the system, the number of migrations, the number of times the scheduler mechanism has been invoked and the number of times it succeeds or fails to resolve non-viable configurations. The TRACE API is extensible in that as many variables as necessary can be created by users of our system, thus allowing researchers to instrument their own algorithm with specific variables that record other pieces of information.

4 Dynamic VMPP Algorithms

To illustrate the interest of VMPlaceS, we implemented three dynamic VM placement mechanisms: a centralized one based on the Entropy proposal [**?**], a hierarchical one based on Snooze [**?**], and a fully-distributed one based on DVMS [**?**]. These systems search for solutions to violations caused by overloaded nodes. A host is overloaded when its VMs try to consume more than 100% of the CPU capacity of the host. In such a case, a resolution algorithm looks for a reconfiguration plan that can lead to a viable configuration. For the sake of simplicity, we chose to use the latest solver developed as part of the Entropy framework [**?**] as this resolution algorithm for all three systems. The Entropy solver evaluates different viable configurations until it reaches a predefined timeout. Once the timeout has been triggered, the algorithm returns the best solution among the ones it finds and applies the associated reconfiguration plan by invoking live migrations in the simulation world.

In the remainder of this section, we present an overview of the three systems.

Entropy-based Centralized Approach. The centralized placement mechanism consists in one single SimGrid process deployed on a service node. This process implements a simple loop that iteratively checks the viability of the current configuration by invoking the aforementioned VMPP solver with a predefined frequency. The resource usage is monitored through direct accesses to the states of the hosts and their respective VMs. We also monitor, for each iteration, whether the VMPP solver succeeds or fails. In the case of success, VMPlaceS records the number of migrations that have been performed, the time it took to apply the reconfiguration and whether the reconfiguration led to new violations.

Snooze-based Hierarchical Approach. Snooze [**?**,**?**] harnesses a hierarchical architecture in order to support load balancing and fault tolerance, cf. Fig. **??**. At the top, a *group leader (GL)* centralizes information about the whole cluster using summary data about *group managers (GMs)* that constitute the intermediate layer of the hierarchy. GMs manage a number of *local controllers (LCs)* that, in turn, manage the VMs assigned to nodes.

[1] http://simgrid.gforge.inria.fr/simgrid/3.12/doc/tracing.html

During execution, higher-level components periodically send heartbeats to lower-level ones; monitoring information, *e.g.*, about the system load, is also sent periodically in the opposite direction. In order to propagate information, Snooze relies on hardware support for multicast communication.

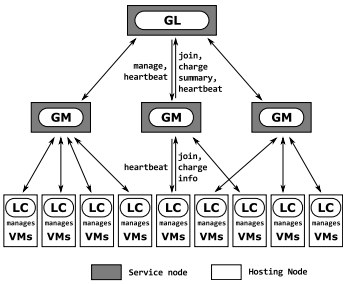


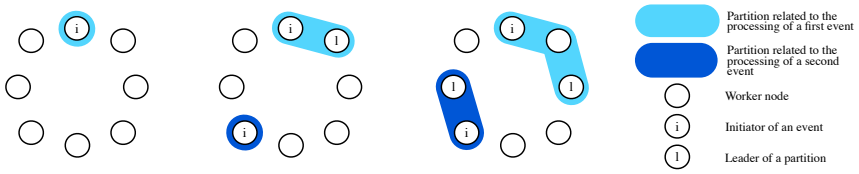
Fig. 2. Snooze Architecture

The implementation in VMPlaceS of the core architectural abstractions of Snooze leverages the `XHOST`, `XVM` and `SimulatorManager` while other mechanisms have been implemented using Simgrid's primitives and standard Java mechanisms. For instance, communication between Snooze actors is implemented based on Simgrid's primitives for, mainly asynchronous, event handling. The multicast capability that is used, *e.g.*, to relay heartbeats, is implemented as a dedicated service that manages a state to relay heartbeat events in a concurrent manner to all receivers. Finally, our Snooze simulation uses, as its original counterpart, a multi-threaded implementation (*i.e.*, based on multiple SimGrid processes) in order to optimize reactivity even for large groups of LCs (or GMs) that have to be managed by one GM (or GL).

DVMS-based Distributed Approach. DVMS (Distributed Virtual Machine Scheduler) [**?**] enables the cooperative and fully-distributed placement of VMs. A DVMS agent is deployed on each node in order to manage the VMs on the node and collaborate with (the agents of) neighboring nodes. Agents are defined on top of an overlay communication network that defines the node-neighbor relation. We have implemented a simple unstructured overlay that enables the agents to collaborate by providing a link to a neighbor on the latter's request.

Fig. **??** depicts the DVMS algorithm. When a node N_i detects that it cannot provide enough resources for its hosted VMs, an *Iterative Scheduling Procedure (ISP)* is started: it initiates a partition, reserving itself to solve the problem, see Fig. **??**. Then, its closest neighbor is considered. If this neighbor, N_{i+1}, is already part of another partition, the next neighbor is considered. Otherwise, N_{i+1} joins the partition (see Fig. **??**) and becomes the partition leader.

The other nodes involved in the partition then send it information about their capacities and current load. The leader, in turn, starts a scheduling computation looking for a reconfiguration within the current partition. If no solution is found, the same algorithm is applied to the next node N_{i+2}. This approach constructs small partitions in a highly parallel manner (Fig. **??**), thus accelerating the scheduling process and reactivity.



(a) (b) (c) (d) Legend

Fig. 3. Processing two events simultaneously

Most of the DVMS code has been coded in SCALA leveraging the Java primitives of SimGrid for the communications between the different DVMS agents that have been implemented, in turn, using the abstractions of VMPlaceS.

5 Experiments

Two kinds of experiments have been performed to validate the relevance of VMPlaceS. The objective of the first one was to evaluate the accuracy of the returned results while the second was a concrete use-case of VMPlaceS, analyzing the three strategies introduced before.

5.1 Accuracy Evaluation

To validate the accuracy of VMPlaceS, we have implemented a dedicated version of our framework[2] on top of the Grid'5000 testbed and compared the execution of the Entropy strategy invoked every 60 seconds over a 3600 seconds period in both the simulated and the real world. Regarding the *in-vivo* conditions, experiments have been performed on top of the Graphene cluster (Intel Xeon X3440-4 CPU cores, 16 GB memory, a GbE NIC, Linux 3.2, Qemu 1.5 and SFQ network policy enabled) with 6 VMs per node. Each VM has been created using one of 8 VM predefined classes. The template was 1:1GB:1Gbps:1Gbps:X, where the memory update speed X was a value between 0 and 80% of the migration bandwidth (1Gbps) in steps of 10. Starting from 0%, the load of each VM varied according to the exponential and the Gaussian distributions. The parameters were λ = #VMs/300 and μ= 60, σ = 20. Concretely, the load of each VM varied on average every 5 min in steps of 10 (with a significant part between 40% and 80%). A dedicated `memtouch` program [**?**] has been used to stress both the CPU and the memory accordingly. Regarding the simulated executions, VMPlaceS has been configured to reflect the *in-vivo* conditions. In particular, we configured the network model of SimGrid in order to cope with the network performance of the Graphene servers that were allocated to our experiment (6 MBytes for the TCP gamma parameter and 0.88 for the bandwidth corrective simulation factor). Fig. **??** shows the time to perform the two phases of the Entropy algorithm for each invocation when considering 32 PMs and 192 VMs through simulations (top) and in reality (bottom). Overall, we can see that simulation results successfully followed the in-vivo ones. During the first hundreds seconds, the cluster did not experience VM requirement violations because

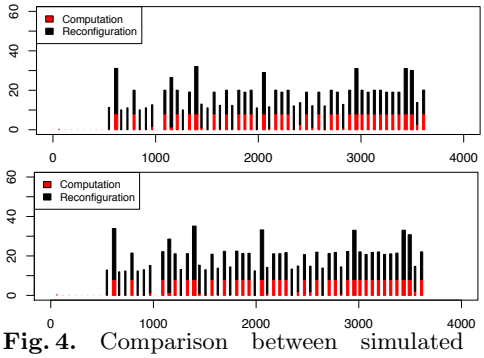


Fig. 4. Comparison between simulated (top) and *in-vivo* (bottom) Executions
The Y-axis represents the duration of each Entropy invocation. It is divived into two parts: the time to look for a new configuration (the computation phase in red) and the time to relocate the VMs (the reconfugration phase in black). Both axis are in seconds.

[2] https://github.com/BeyondTheClouds/G5K-VMPlaceS

the loads of VM were still small (*i.e.*,
Entropy simply validated that the current placement satisfied all VM requirements). At 540 seconds, Entropy started to detect non viable configurations and performed reconfigurations. Diving into details, the difference between the *simulated* and *in-vivo* reconfiguration time fluctuated between 6% and 18% (median was around 12%). The worst case, *i.e.*, 18%, was reached when multiple migrations were performed simultaneously on the same destination node. In this case and even if the SFQ network policy was enabled, we discovered that in the reality the throughput of migration traffic fluctuated when multiple migration sessions simultaneously shared the same destination node. We confirmed this point by analyzing TCP bandwidth sharing through `iperf` executions. We are currently investigating with the SimGrid core-developers how we can integrate this phenomenon into the live-migration model. However, as a migration lasts less than 15 seconds in average, we believe that that the current simulation results are sufficiently accurate to capture performance trends of placement strategies.

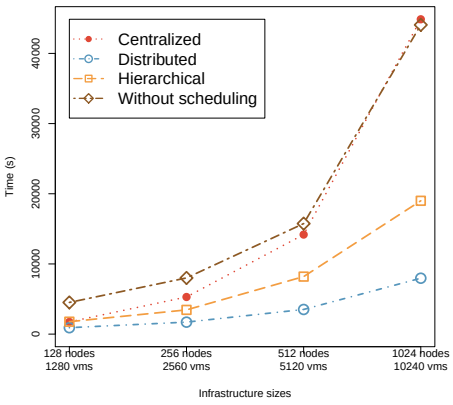
5.2 Analysis of Entropy, Snooze and DVMS

As a validation of our approach (and a contribution by itself), we now provide simulation results comparing the Entropy, Snooze and DVMS strategies.

Experimental Conditions. Each simulation has been executed on a dedicated server, thus avoiding interferences between simulations and ensuring reproducibility between the different invocations. VMPlaceS has been configured to simulate a homogeneous infrastructure of PMs composed of 8 cores, 32 GB of RAM and 1 Gpbs Ethernet NIC. To enable a fair comparison between the three strategies, the scheduling resolver only considered 7 cores, *i.e.*, one was devoted to run the Snooze LC or the DVMS admin processes (a common experimental setup). Ten VMs have been initially launched on each simulated PM. Each VM relied on one of the VM classes described in the accuracy experiment and one set of load-change parameters has been used: $\lambda = \#\text{VMs}/300$, $\mu = 60$ and $\sigma = 20$. The stationary state was reached after 20 min of the simulated time with a global cluster load of 85%. We have performed simulations over a period of 1800 seconds. The consolidation ratio, *i.e.*, the number of VMs per node, has been defined such that a sufficient number of violations is generated. We have discovered that below a global load of 75%, few VM violations occurred under the selected Gaussian distribution we have chosen. This result is rather satisfactory as it can explained why most production DCs target a comparable load level.[3] Finally, infrastructures composed of 128, 256, 512 and 1024 PMs, hosting respectively 1280, 2560, 5120 and 10240 VMs have been investigated. For Entropy and Snooze that rely on service nodes, additional simulated PMs have been provided. For Snooze, one GM has been created per 32 LCs (*i.e.*, PMs). The solver has been invoked every 30s for Entropy and Snooze.

General Analysis. Figure **??** presents on the left the cumulated violation time for each placement policy and on the right several tables that give more de-

[3] http://www.cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually/



Infrastructure size	Duration of **violations** ($\mu \pm \sigma$)		
	Centralized	Hierarchical	Distributed
128 nodes	21.26 ± 13.55	21.07 ± 12.32	9.55 ± 2.57
256 nodes	40.09 ± 24.15	21.45 ± 12.10	9.58 ± 2.51
512 nodes	55.63 ± 42.26	24.54 ± 16.95	9.57 ± 2.67
1024 nodes	81.57 ± 86.59	29.01 ± 38.14	9.61 ± 2.54

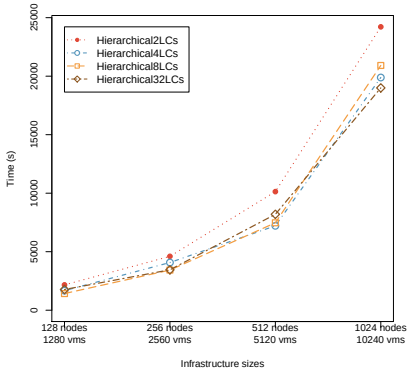
Infrastructure size	Duration of **computations** ($\mu \pm \sigma$)		
	Centralized	Hierarchical	Distributed
128 nodes	3.76 ± 7.43	2.52 ± 4.63	0.29 ± 0.03
256 nodes	7.97 ± 15.03	2.65 ± 4.69	0.25 ± 0.02
512 nodes	15.71 ± 29.14	2.83 ± 4.98	0.21 ± 0.01
1024 nodes	26.41 ± 50.35	2.69 ± 4.92	0.14 ± 0.01

Infrastructure size	Duration of **reconfigurations** ($\mu \pm \sigma$)		
	Centralized	Hierarchical	Distributed
128 nodes	10.34 ± 1.70	10.02 ± 0.14	10.01 ± 0.11
256 nodes	10.26 ± 1.45	10.11 ± 0.83	10.01 ± 0.08
512 nodes	11.11 ± 3.23	10.28 ± 1.50	10.08 ± 0.82
1024 nodes	18.90 ± 7.57	10.30 ± 1.60	10.04 ± 0.63

Fig. 5. Scalability/Reactivity analysis of Entropy, Snooze and DVMS

tails by presenting the mean and the standard deviations of the duration of, respectively, the violations and the computation/reconfiguration phases. As anticipated, the centralized approach did not scale and even incurs an overhead in the largest scenario compared to a system that did not perform any dynamic scheduling. The more nodes Entropy has to monitor, the less efficient it is during both the computation and reconfiguration phases. This is to be expected for the computation phase (which tries to tackle an NP-complete problem). As to reconfiguration, the reconfiguration plan becomes more complex for large scenarios, including several migrations coming from and going to the same nodes. Such plans are not optimal as they increase the bottleneck effects at the network level of each involved PM. Such a simulated result is valuable as it confirms that reconfiguration plans should avoid such manipulations as much as possible. The results of the hierarchical approach are clearly better than the Entropy-based ones but worse than those using DVMS-based placement. However, diving into the details, we can see that both the time needed for the computation and reconfiguration are almost independent from the cluster size (around 3s and 10s) and not much worse than those of DVMS, especially for the reconfiguration phase, which is predominant. These results can be easily explained: the centralized policy addresses the VMPP by considering all nodes at each invocation, while the hierarchical and the distributed algorithms divide the VMPP into sub problems, considering smaller numbers of nodes (32 PMs in Snooze and, on average, 4 in the case of DVMS). To clarify the influence of the group size on the performance of Snooze, *i.e.*, the ratio of LCs attached to one GM, we have performed additional simulations for varying group sizes. VMPlaceS has significantly facilitated this study as the corresponding simulations differ just by configuration parameters and do not require modifications to the code base.

Investigating Algorithm Variants. VMPlaceS facilitates the in-depth analysis of variants of placement algorithms. We have, for example, analyzed, as a first study of its kind, how the Snooze-based placement depends on the no. of LCs assigned to a GM. Fig. **??** presents the simulated values obtained for sce-



Infra. Size	No. of **failed reconfigurations**			
	2 LCs	4 LCs	8 LCs	32 LCs
128	19	0	0	0
256	29	0	0	0
512	83	1	0	0
1024	173	7	0	0

Infra. Size	Duration of the **computations** ($\mu \pm \sigma$)			
	2 LCs	4 LCs	8 LCs	32 LCs
128	0.16 ± 1.23	0.34 ± 1.81	0.58 ± 2.40	2.53 ± 4.62
256	0.18 ± 1.31	0.42 ± 1.99	0.66 ± 2.50	2.65 ± 4.69
512	0.15 ± 1.20	0.33 ± 1.78	0.67 ± 2.54	2.83 ± 4.98
1024	0.19 ± 1.37	0.42 ± 2.02	0.89 ± 2.90	2.69 ± 4.91

Fig. 6. Hierarchical placement: influence of varying group sizes

narios with 2, 4, 8 and 32 LCs per GM for four infrastructure sizes. The overall performance (*i.e.*, cumulated violation time) shows that 2 LCs per GM result in significantly higher violation times. The relatively bad performance of the smallest group size can be explained in terms of the number of failures of the reconfiguration process, that is, overloading situations that are discovered but cannot be resolved due to a lack of resources (see tables on the right). Groups of 2 LCs per GM are clearly insufficient at our global load level (85%). Failed reconfigurations are, however, already very rare in the case of 4 LCs per GM and do not occur at all for 8 and 32 LCs per GM. This is understandable because the load profile we evaluated rarely results in many LCs of a GM to be overloaded at once. Violations can therefore be resolved even in the case of a smaller number of LCs available for load distribution. Conversely, we can see that the duration of the computation phases decreases strongly along with the group size. It reaches a value close to the computation times of DVMS for a group size of 4-LCs per GM.We thus cannot minimize computation times and violation times by reducing the number of LCs because larger group sizes are necessary to resolve overload situations if the VM load gets higher. In contrast, DVMS resolves this trade-off by means of its automatic and dynamic choice of the partition size necessary to handle an overload situation. Once again, this information is valuable as it will help researchers to design new algorithms favoring the automatic discovery of the optimal subset of nodes capable to solve violations for given load profiles.

The study performed in this paper has allowed us to analyze several other variants and possible improvements (which we cannot present here for lack of space), such as a reactive approach to hierarchical placement instead of the periodical one used by Snooze, as well as more aggressive partitioning in the case of DVMS. VMPlaceS also provides additional metrics such as the overall count of migrations, the average duration of each migration ... These allow important properties, *e.g.*, the migration overhead, to be studied. All these variants can be easily studied and evaluated thanks to VMPlaceS.

Finally, we have succeeded to conduct DVMS simulations up to 8K PMs/80K VMs in a bit less than two days. We did not present these results in this paper because it was not possible to run a sufficient number of

Snooze simulations at such a scale (the Snooze protocol being more complex). The time-consuming portions of the code are related to SimGrid internals such as `sleep` and `send/recv` calls. Hence, we are collaborating with SimGrid core developers in order to reduce the simulation time in such cases.

6 Related Work

Simulator toolkits that have been proposed to address CC concerns [**?**,**?**,**?**,**?**,**?**] can be classified into two categories. The first corresponds to ad-hoc simulators that have been developed to address one particular concern. For instance, CReST [**?**] is a discrete event simulation toolkit built for Cloud provisioning algorithms. If ad-hoc simulators allow some characteristics of the behaviors of the system to be analyzed, they do not consider the implication of the different layers, which can lead to non-representative results. Moreover, most ad-hoc solutions are developed for one shot analyses. That is, there is no effort to release them as a complete and reusable tool for the scientific community. The second category [**?**,**?**,**?**] corresponds to more generic cloud simulator toolkits (*i.e.*, they have been designed to address multiple CC challenges). However, they focus mainly on the API and not on the model of the different mechanisms of CC systems. For instance, CloudSim [**?**], which has been widely used to validate algorithms and applications in different scientific publications, is based on a top-down viewpoint of cloud environments. That is, there are no articles that properly validate the different models it relies on: a migration time is simply (and often imprecisely) calculated by dividing VM memory sizes by network bandwidth values. In addition to be subject to inaccuracies at the low level, available cloud simulator toolkits often use oversimplified models for virtualization technologies, also leading to non-representative results. As highlighted throughout this article, we have chosen to build VMPlaceS on top of SimGrid in order to build a generic tool that benefits from the accuracy of its models related to virtualization abstractions [**?**].

7 Conclusion

We have presented VMPlaceS, a framework providing generic programming support for the definition of VM placement algorithms, execution support for their simulation at large scales, as well as new means for their trace-based analysis. We have validated its accuracy by comparing simulated and *in-vivo* executions of the Entropy strategy. We have also illustrated the relevance of VMPlaceS by evaluating and comparing algorithms representative of three different classes of virtualization environments: centralized, hierarchical and fully distributed placement algorithms. The corresponding experiments have provided the first systematic results comparing these algorithms in environments including up to one 1K nodes and 10K VMs.

A version of VMPlaceS is available on a public git repository[4]. We are in touch with the SimGrid core developers in order to improve our code with the ultimate objective of addressing infrastructures up to 100K PMs and 1 Millions

[4] http://beyondtheclouds.github.io/VMPlaceS/

VMs. As future work, it would be valuable to add additional dimensions in order to simulate other workload variations stemming from network and HDD I/O changes. Moreover, we plan to provide a dedicated API to be able to provision and remove VMs during the execution of a simulation.