SimGrid VM: Virtual Machine Support for a Simulation Framework of Distributed Systems

Takahiro Hirofuchi, Adrien Lebre, and Laurent Pouilloux

Abstract—As real systems become larger and more complex, the use of simulator frameworks grows in our research community. By leveraging them, users can focus on the major aspects of their algorithm, run *in-siclo* experiments (*i.e.,* simulations), and thoroughly analyze results, even for a large-scale environment without facing the complexity of conducting in-vivo studies (*i.e.,* on real testbeds). Since nowadays the virtual machine (VM) technology has become a fundamental building block of distributed computing environments, in particular in cloud infrastructures, our community needs a full-fledged simulation framework that enables us to investigate large-scale virtualized environments through accurate simulations. To be adopted, such a framework should provide easy-to-use APIs as well as accurate simulation results. In this paper, we present a highly-scalable and versatile simulation framework supporting VM environments. By leveraging SimGrid, a widely-used open-source simulation toolkit, our simulation framework allows users to launch hundreds of thousands of VMs on their simulation programs and control VMs in the same manner as in the real world (e.g., suspend/resume and migrate). Users can execute computation and communication tasks on physical machines (PMs) and VMs through the same SimGrid API, which will provide a seamless migration path to IaaS simulations for hundreds of SimGrid users. Moreover, SimGrid VM includes a live migration model implementing the precopy migration algorithm. This model correctly calculates the migration time as well as the migration traffic, taking account of resource contention caused by other computations and data exchanges within the whole system. This allows user to obtain accurate results of dynamic virtualized systems. We confirmed accuracy of both the VM and the live migration models by conducting several micro-benchmarks under various conditions. Finally, we conclude the article by presenting a first use-case of one consolidation algorithm dealing with a significant number of VMs/PMs. In addition to confirming the accuracy and scalability of our framework, this first scenario illustrates the main interest of SimGrid VM: investigating through in-siclo experiments pros/cons of new algorithms in order to limit expensive in-vivo experiments only to the most promising ones.

Index Terms—Virtual Machine; Simulation; Cloud Computing; Live Migration;

1 Introduction

NOWADAYS, virtual machine (VM) technology plays one of the key roles in cloud computing environments. Large-scale data centers can manipulate up to one million of VMs, each of them being dynamically created and destroyed according to user requests. Numerous studies on large-scale virtualized environments are being conducted by both academics and industries in order to improve performance and reliability of such systems. However, these studies sometimes involve a potential pitfall; the number of virtual machine (VMs) considered to validate research proposals (up to one thousand in the best case) is far less than the number actually hosted by real Infrastructure-as-a-Service (IaaS) platforms. Such a gap prevents researchers from corroborating the relevance of managing VMs in a more dynamic fashion, since valid assumptions on small infrastructures sometimes become completely erroneous on much larger ones. The reason behind this pitfall is that it is not always possible for researchers to evaluate robustness and performance of cloud computing platforms through large-scale "in vivo" (*i.e.*, real-world) experiments. In addition to the difficulty in obtaining an appropriate testbed, it implies expensive and tedious tasks such as controlling, monitoring and ensuring the reproducibility of the experiments. Hence, correctness of most contributions in the management of IaaS platforms has been proved by means of ad-hoc simulators and confirmed, when available, with small-scale "in vivo" experiments. Even though such approaches enable our community to make a certain degree of progress, we advocate that leveraging ad-hoc simulators with not so representative "in vivo" experiments is not rigorous enough to compare and validate new proposals.

Like for Grids [1], P2P [2] and more recently highest levels of cloud systems [3], the IaaS community needs an open simulation framework for evaluating concerns related to the management of VMs. Such a framework should allow investigating very large-scale simulations in an efficient and accurate way as well as it should provide adequate analysis tools to make the discussions of results as clear as possible: By leveraging simulation results, researchers will be able to limit "in vivo" experiments only to the most relevant ones.

In this paper, we present SimGrid VM, the first highly-scalable and versatile simulation framework supporting VM environments. We chose to build it upon SimGrid [1] since its relevance in terms of performance and validity has already been demonstrated for many distributed systems [4].

Our simulation framework allows users to launch hundreds of thousands of VMs on their simulation programs and accurately control VMs in the same manner as in the real world. The framework correctly calculates resource allocation to each computation/communication

- *Takahiro Hirofuchi is Researcher at AIST, Japan Email: t.hirofuchi at aist.go.jp.*
- *Adrien Lebre and Laurent Pouilloux are respectively Researcher and Research Engineer at Inria, France Email: {adrien.lebre,laurent.pouilloux} at inria.fr.*

This work is supported by the French ANR project SONGS (11-INFRA-13) and the JSPS KAKENHI Grant 25330097.

task, considering VM placement on a simulated system.

The extension to SimGrid has been designed and implemented in order to be as transparent as possible to other components: SimGrid users can now easily manipulate VMs while continuing to take the advantage of the usual SimGrid MSG API for creating computation and communication tasks either on PMs or on VMs. Such a choice provides a seamless migration path from traditional clusters simulations to IaaS ones for hundreds of SimGrid users. This has been made possible by introducing the concept of abstraction level (i.e., physical and virtual) into the core of SimGrid. A virtual workstation model, inheriting from the workstation one, has been added into the framework and overrides only the operations that are VM-specifics.

In addition to the virtual workstation model, we also integrated a live migration model implementing the precopy migration algorithm. This model correctly calculates the migration time as well as the migration traffic, taking account of resource contention caused by other computations and data exchanges within the whole system. This allows user to obtain accurate results of systems where migrations play a major role. This is an important contribution as several people might erroneously consider that live migrations can be simulated by simply leveraging data transfer models. As discussed in this article, several parameters such as the memory update speed of a VM govern live migration operations and should be considered in the model if we want to deliver correct values.

The rest of the paper is organized as follow. Sec. 2 gives an overview of SimGrid. After summarizing the requirements for the VM support in a simulation framework, Sec. 3 introduces SimGrid VM and explains in details how the virtual workstation and the live migration models have been designed. Evaluation of both models is discussed in Sec. 4. A first use-case of the VM extensions is presented in Sec. 5. Sec. 6 deals with related work and finally, Sec. 7 concludes and gives some perspectives for SimGrid VM.

2 SimGrid Overview

SimGrid is a simulation framework to study the behavior of large-scale distributed systems such as Grids, HPC and P2P systems. There is a large, world-wide user community of SimGrid. The design overview of SimGrid was described in [1]. SimGrid is carefully designed to be scalable and extensible. It is possible to run a simulation composed of 2,000,000 processors on a computer with 16GB of memory. It allows running a simulation on arbitrary network topology under dynamic compute and network resource availabilities. It allows users to quickly develop a simulating program through easy-to-use APIs in C and Java.

Users can dynamically create two types of tasks, i.e., computation tasks and communication tasks, in a simulation world. As the simulation clock is going forward, these tasks are being executed, consuming CPU and network resource in the simulation world. Behind the scene, SimGrid formalizes constraint problems[1] to get a resource share to each task. Even though there is complex resource contention, it can formulate constraint problems expressing such a situation. Until the simulation ends, it repeats formalizing constraint problems, solving them, and then continuing with the next simulation step. As input parameters, users will prepare *platform* files that describe how their simulation environments are organized; for example, they will include the CPU capacity of each host and the network topology of their environments.

Although SimGrid has many features such as model checking, the simulation of MPI applications, and the task scheduling simulation of DAGs (Direct Acyclic Graphs), we limit our description to the fundamental parts, which are directly related to the VM support.

SimGrid is composed of three different layers:

The MSG layer provides programming APIs for users. In most cases, users develop their simulation programs only using the APIs in this layer. It allows users to create a process on a host and to execute a computation/communication task on it.

The SIMIX layer is located in the middle of the components. This layer works for the synchronization and scheduling of process executions on a simulation. Roughly, it provides a functionality similar to system calls in the real world, i.e., *simcall* in the SimGrid terminology. When a process calls a simcall to do an operation (e.g., compute, or send/receive data) in the simulation world, the SIMIX layer converts it to an action in a corresponding simulation model (explained later). Then, the SIMIX layer blocks the execution of the process until the operation is completed in the simulation world. If there is another process that is not yet blocked, the SIMIX layer performs the context switch to another process, converts its operation to another action, and blocks the execution of that process. After all processes are blocked (i.e., the SIMIX layer has converted the operations of all the currently-running processes to actions), the SIMIX layer requests each model to solve constraint problems, which determines the resource share of each action. After all constraint problems are solved, the SIMIX layer sets the simulation clock ahead until at least one action is completed under the determined resource shares. Then, it restarts the execution of the processes of the completed actions. These steps are repeated until a simulation is over.

The SURF layer is the kernel of SimGrid, where a simulation model of each resource is implemented. A model formulates constraint problems according to requests from the SIMIX layer, and then actually solves them to get the resource share of each action. There are a CPU model and a network model, used for computation and communication tasks, respectively.

In addition to the MSG API, SimGrid provides several abstractions such as the TRACE mechanisms that enable

1. Constraint problem, or constraint satisfaction problem, is a mathematical problem in which equations define requirements that variables should meet.

to perform fine post-mortem analysis of the users' simulations with some visualization tools like the Triva/Viva/PajeNG software suite [5]. SimGrid will produce rich simulation outputs allowing users to perform statistical analysis on simulation results.

3 The VM Support in SimGrid

The extension of SimGrid to support VM abstractions has been driven by the following requirements:

- Produce accurate results. Our simulation framework requires the capability to accurately determine resource shares on both virtualized and non-virtualized systems, which must take account of VM placement on physical resources as well as the mechanism of virtualization. In other words, if VMs are co-located on a PM, the system should calculate a correct CPU time to each VM. If a network link is shared with multiple data transfers from/to VMs or PMs, the system needs to assign a correct bandwidth to each transfer.
- Achieve high scalability. SimGrid has been carefully designed to be able to perform large-scale simulations. The VM support on SimGrid must also achieve high scalability for large simulation environments comprising a mix of thousands of VMs and PMs.

After giving an overview of how we introduced the concept of virtual and physical machine layers in its solving engine of constraint problems, we present the live migration model we implemented. Thanks to these extensions, users can now simulate the computation and the communication of virtualized environments as well as investigating mechanisms involving VM live migration operations.

3.1 Adding a VM Workstation Model

In the extension for the VM support, we introduced the concept of an abstraction level in the core of SimGrid, i.e., the PM level and VM level. This design enables us to leverage the constraint problem solver of SimGrid also for the VM support. No modification to the solver engine has been required.

Fig. 1 illustrates the design overview of SimGrid with the VM support. We added the virtual workstation model to the SURF layer, and also modified the SIMIX layer to be independent of the physical and virtual workstations. A workstation model is responsible for managing resources in the PM or VM layer. The virtual workstation model inherits most callbacks of the physical one, but implementing VM-specific callbacks. When a process requests to execute a new computation task, the SIMIX layer calls the SURF API of the corresponding workstation model (i.e. depending on where the task is running, the PM or a VM workstation model is used). Then, the target workstation model creates a computation action, and adds a new constraint problem into that layer.

In the SURF layer, the physical workstation model creates PM resource objects for each simulated PM. A

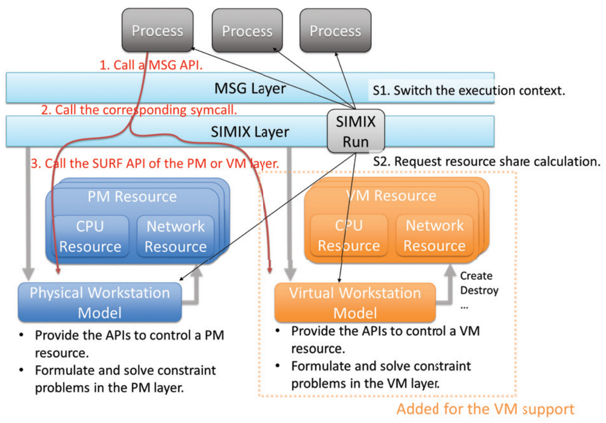

Process
Process
Process
1. Call a MSG API.
MSG Layer
S1. Switch the execution context.
2. Call the corresponding symcall.
SIMIX Layer
SIMIX Run
3. Call the SURF API of the PM or VM layer.
S2. Request resource share calculation.
PM Resource
CPU Resource
Network Resource
VM Resource
CPU Resource
Network Resource
Create
Destroy
...
Physical Workstation Model
• Provide the APIs to control a PM resource.
• Formulate and solve constraint problems in the PM layer.
Virtual Workstation Model
• Provide the APIs to control a VM resource.
• Formulate and solve constraint problems in the VM layer.
Added for the VM support

Fig. 1: Design Overview of the VM Support in SimGrid

During a simulation, SimGrid repeats the following steps: (S1) The SIMIX Run function switches the execution context to each process. (1-2-3) Each process calls a MSG API function, and the corresponding workstation model formulates constraint problems. (S2) The SIMIX Run function requests to solve constraint problems. Then, it updates the states of processes with solved values and sets the simulation clock ahead.

PM resource object is composed of a CPU resource object and a network resource object. A CPU resource object has the capability (flop/s, floating operation per second) of the PM. A network resource object corresponds to the network interface of the PM. A VM resource object basically has the same structure as a PM one, including a CPU resource object and a network object. However, a VM resource object has the pointer to the PM resource object where the VM is running. It also has a dummy computation action, which represents the CPU share of the VM in the PM layer (i.e., a variable object X_i in the following). Currently, we mainly focus on the simulation of CPU and network resources. Disk resource simulation will be integrated in the near future.

As explained in Sec. 2, the SIMIX RUN function executes processes in a simulation in a one-by-one fashion, and then requests each workstation model to calculate resource shares in each machine layer. We modified the SIMIX RUN function to be aware of the machine layers on the simulation system. In theory, it is possible to support the simulation of nested virtualization (i.e., execute a VM at the inside of another VM) by adding another machine layer to the code.

The extension of the VM support solves constraint problems with 2 steps. First, the system solves the constraint problems in the PM layer, and obtains the values of how much resource is assigned to each VM (using the corresponding dummy action of each VM). Then, the system solves the constraint problems in the VM layer. From the viewpoint of a PM, a VM is considered as an ordinary task on the PM. From the viewpoint of a task inside a VM, a VM is considered as an ordinary host below the task.

Without the VM support, the solver engine solves all constraint problems on a simulation at once. The left side of Fig. 2 shows a simple example where 2 computation tasks are executed on a PM. The PM has a CPU of the capacity C (flop/s). Then, the system formulates a

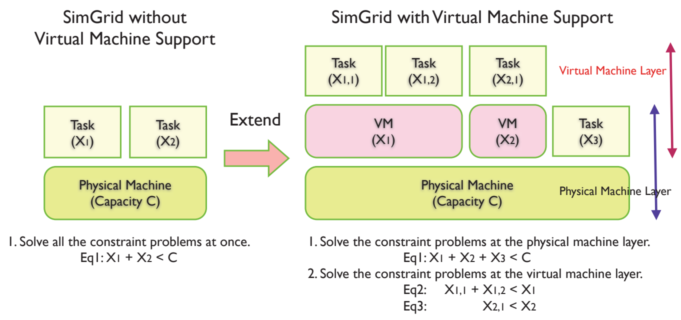

SimGrid without
Virtual Machine Support
Task (X1)
Task (X2)
Physical Machine (Capacity C)
Extend
SimGrid with Virtual Machine Support
Task (X1,1)
Task (X1,2)
Task (X2,1)
Virtual Machine Layer
VM (X1)
VM (X2)
Task (X3)
Physical Machine (Capacity C)
Physical Machine Layer
1. Solve all the constraint problems at once.
Eq1: X1 + X2 < C
1. Solve the constraint problems at the physical machine layer.
Eq1: X1 + X2 + X3 < C
2. Solve the constraint problems at the virtual machine layer.
Eq2: X1,1 + X1,2 < X1
Eq3: X2,1 < X2

Fig. 2: Resource Share Calculation with VM Support

C is the CPU capacity (*i.e.*, the processing speed in flop/s) of a PM. X_i are CPU resource shares assigned to tasks or VMs at the PM layer (from the Host OS viewpoint, a VM is regarded as a task). $X_{i,j}$ are those of the tasks running inside the VM_i.

constraint problem,

$$X_1 + X_2 < C \tag{1}$$

where X_1 and X_2 are the CPU shares of each task, respectively. If there are no other conditions, the solver engine assigns 50% of the computation capacity of the PM to each task.

The right side of Fig. 2 shows an example with the VM support. A PM has two VMs (VM1 and VM2) and a computation task. VM1 has two computation tasks, and VM2 has a computation task. First, the system formulates a constraint problem at the PM layer.

$$X_1 + X_2 + X_3 < C \tag{2}$$

where X_1, X_2, and X_3 are the CPU shares of VM1, VM2, and the task on the PM. If there are no other conditions, the solver engine assigns 33.3% of the computation capacity of the PM to VM1, VM2 and the task on the PM. Second, the system formulates a constraint problem at the VM layer. Regarding VM1 executing 2 computation tasks, the solver engine makes

$$X_{1,1} + X_{1,2} < X_1 \tag{3}$$

where $X_{1,1}$, $X_{1,2}$ are the CPU shares of the tasks on VM1. In the same manner, for VM2 executing 1 computation task, the solver engine makes

$$X_{2,1} < X_2 \tag{4}$$

where $X_{2,1}$ is the CPU shares of the task on VM2. Thus, if there are no other conditions, each task on VM1 obtains 16.7% of the CPU capacity while the VM2 one obtains 33.3%.

SimGrid allows end-users to set priority to each task. This capability also works for the VM support. As for the above example, if we set 2x larger priority to VM1, VM1 obtains 50% of the computation capacity, and VM2 and the task on the PM get only 25%, respectively. Additionally, we added the capping mechanism of the maximum CPU utilization of each task and VM. We can set the maximum CPU utilization of VM1 to 10% of the capacity of the PM. Even if we remove VM2 and the task on the PM, VM1 cannot obtain more than 10%. These features are useful to take account of virtualization overheads in simulations. In the real world, we can sometimes observe that the performance of a workload is degraded at the inside of a VM. It is possible to simulate this kind of overhead by means of setting priority and capping of tasks and VMs appropriately.

The network resource calculation mechanism with the VM support is implemented in the same manner as the CPU mechanism. The network mechanism considers the resource contention on the PM and also that of shared network links.

3.2 Adding a Live Migration Model

Virtual machine monitors (VMMs) supporting live migration of VMs usually implement the precopy algorithm [6]. At coarse grained, this algorithm transfers all memory pages to the destination PM, before switching the execution host of a VM. Thus, one can erroneously envision that live migration operations can be simulated simply by one network exchange between the source and the destination nodes. However, the pre-copy algorithm is a bit more complex and it is crucial to consider several parameters that clearly govern the time that is mandatory to migrate one VM from one node to another. In this section, first, we describe the successive steps of the precopy algorithm and show that the memory update speed of the VM governs this algorithm by discussing several micro-benchmarks. The design of our live migration model that relies on this preliminary study is finally introduced.

3.2.1 Live Migration Fundamentals

When a live migration is invoked for a particular VM, the VMM performs the following stages:

- Stage 1: Transfer all memory pages of the VM. Note that the guest operating system is still running at the source. Hence, some memory pages can be updated during this first transfer.
- Stage 2: Transfer the memory pages that have been updated during the previous copy phase. Similar to Stage 1, some memory pages will be updated during this second transfer. Hence Stage 2 is iteratively performed until the number of updated memory pages becomes sufficiently small.
- Stage 3: Stop the VM. Transfer the rest of memory pages and other negligibly small states (*e.g.*, those of virtual CPU and devices). Finally, restart the VM on the destination.

Since Stage 3 involves a temporal pause of the VM, the VMM tries to minimize this downtime as much as possible, making it unnoticeable from users and applications. For example, the default maximum downtime of Qemu/KVM, the de-fact Linux VMM [7], is 30 ms. During Stage 2, Qemu/KVM iteratively copies updated memory pages to destination, until the size of remaining memory pages becomes smaller than the threshold value that will achieve the 30 ms downtime. Consequently, a migration time mainly depends on the memory update speed of the VM and the network speed of the migration traffic. A VM intensively updating memory pages will require a longer migration time and in the worst case (*i.e.*, the memory update speed is higher than the network bandwidth), a migration will not finish (i.e., not *converge* in technical terms). Although libvirt [8] allows users to set a timeout value for a migration, this mechanism is not enabled in the default settings of Linux distributions.

3.2.2 *Impact of The Memory Update Speed*

In order to have an idea of the magnitude of the memory update speed in cloud applications, we performed preliminary experiments using representative workloads. This clarifies whether the memory update speed is large enough to be considered as a predominant factor of a live migration model. First, we used a web server workload, and second a database server. We measured how the memory update speed changes in response to the CPU load change of the VM. Each workload runs on the guest OS of a VM.

To measure the memory update speeds, we extended Qemu/KVM to periodically output the current memory update speed of a VM. The VMM has the mechanism to track updated memory pages during a migration, *i.e.*, dirty page tracking. The VMM maintains the bitmap recording updated memory page offsets. With the extension, the VMM enables dirty page tracking: it scans and clears every second the bitmap in order to count up the number of updated pages. It is noteworthy that we did not observe noticeable CPU overhead due to this extension. The recent hardware supports dirty page tracking in the hardware level, and its CPU overhead is substantially small compared to the CPU consumption of a workload.

3.2.2.1 Web Sever: We set up a VM with one VCPU, 1 GB of memory, and one network interface on a PM. The network interface was bridged to the physical network link of GbE. We configured an Apache-2.2 web server on the guest OS of the VM. The Apache server worked in the multi-threads mode, handling each HTTP session with one thread. Another PM was used to launch the *siege* web server benchmark program [9], which randomly retrieves files available on the web server by performing HTTP get requests. Static web contents had been generated in advance on the guest OS. The size of each file was 100 KB (*i.e.*, a typical size of a web content on the Internet) and the total size of the generated web contents was 2 GB (20K files).

In order to investigate the impact of the page cache mechanism upon the update memory speed, we performed two particular cases:

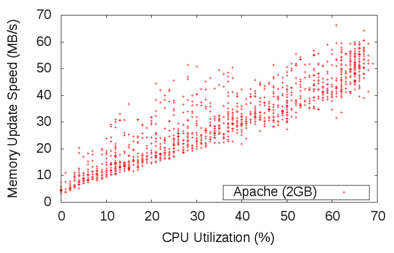
- For the first case, the benchmark program randomly accessed all 2 GB web contents. Because the RAM size of the VM is 1 GB, accessing more than 1 GB involves I/O accesses since the whole contents cannot be cached by the guest OS. When a requested file is not on the page cache, the guest OS reads the content file from the virtual disk, involving memory updates.
- For the second case, we limited HTTP requests only to 512 MB of the web contents. The corresponding 5000 files had been read on the guest OS, before launching the benchmark program. By caching target files beforehand, we minimized memory updates due to the page cache operation.

For both experiments, we gradually increased the number of concurrent accesses performed by the benchmark program: The number of concurrent sessions was increased by 16 every 60 seconds, up to 512. We measured the memory update speed and the CPU utilization every second. Fig. 3a and 3b show the correlation between the CPU utilization level of the VM and its memory update speed. When the number of concurrent sessions increased, the CPU utilization as well as the memory update speed became higher. We expected that the memory update speed of the first case would be significant because of refreshing the page cache, and that of the second case would be small because all file contents were already cached. As shown in Fig. 3b, however, in the second case, there exist intensive memory updates (*e.g.*, 30 MB/s at 60% of the CPU utilization), which are not negligible in comparison to the network bandwidth. Considering that the Apache server sends data through zero copy operations (*e.g.*, the use of `sendpage()`, or the combination of `mmap()` and `send()`), this memory update speed results from the pages used by the web server to manage HTTP session (*i.e.*, the heap on the guest OS). The guest OS kernel will also update memory pages for TCP connections, receiving client requests and sending dynamically generated data (*i.e.*, HTTP protocol headers).

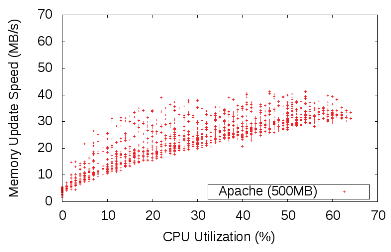
3.2.2.2 Database Server: The second study focuses on a postgresql-9.2.4 database server. The configuration of the VM was the same as that of the web server experiments. The *pgbench* benchmark [10] was used on the other PM. It emulates the TPC-B benchmark specification [11] that targets database management systems on batch applications, and the back-end database server on market segment. The default setting of the benchmark program aims at measuring the maximum performance of a database server and thus completely saturates the CPU utilization of the VM. To observe the behavior of the VM at various CPU utilization levels, we inserted a 20 ms delay at the end of each sequence of transaction queries.

After starting the experiment, we increased the number of concurrent database sessions by 2 every 60 seconds and up to 70 concurrent sessions. Similarly to the Apache experiments, we can observe on Fig. 3c, a clear linear correlation between them. Because every 60 seconds the benchmark program was re-launched with a new concurrency parameter, the sporadic points distant from the majority was caused by establishing new database sessions. As shown in the graph, there exists intensive memory updates of the VM. In this experiment, when the CPU utilization was 60%, the memory update speed reached 40MB/s (30% of the theoretical GbE bandwidth).

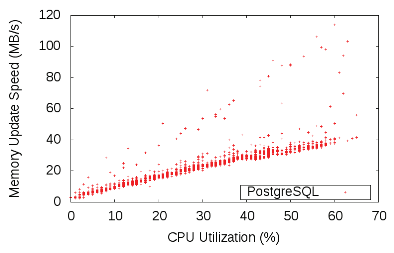
To conclude, although points are scattered in the different graphs, we can observe that a proportional correlation between the CPU usage and the memory update speed exists as a rough trend in the experiments. Such a correlation is important as it will enable to determine the memory update speed of a VM according to its CPU



(a) Apache Web Server

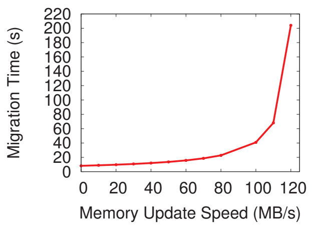


(b) Apache Web Server (cached contents).

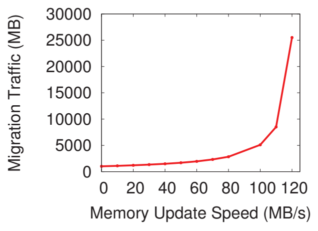


(c) PostgreSQL Database Server

Fig. 3: The correlation between CPU utilization and memory update speed



(a) Migration Time (s)



(b) Migration Traffic (MB)

Fig. 4: Impact of memory updates on migration cost
A VM with 1 GB RAM over a 1 Gbps link. The values are theoretically estimated according to the precopy algorithm.

usage.

3.2.2.3 Summary: Through the above experiments, we have seen that the memory update speed can be quite significant in comparison with the network bandwidth. Providing a naive model, which simply obtains the cost of a live migration by dividing the VM memory size by the available network bandwidth, is not appropriate as the memory update speed determines the duration of Stage 2 of the precopy algorithm. As an example, the naive model will estimate the migration of the aforementioned database to 8 seconds (*i.e.*, the time required for transferring 1 GB data over the 1 Gbps link) for all situations. This might result in a tremendous gap from the migration performance in the real world once the VM starts to be active: when the database VM is utilizing 60% CPU resource, the live migration time of this VM was approximately 12 seconds, and the transferred data size during the migration reached approximately 1500 MB. This corresponds to 1.5 times the migration cost estimated by the naive model. Fig. 4 presents the theoretical estimation of migration cost for a VM (1 GB RAM). The graphs clearly reveal that it is critical to consider the impact of memory updates in order to make an accurate estimation of migration time and generated network traffic.

Moreover, the time to perform the first two stages is easily and greatly affected by activities of other VMs and workloads running in the system:

- From the network point of view, the traffic of a live migration can suffer from other network exchanges. For instance, a workload of a VM may create network traffics that competes with migration ones. Similarly, such a competition occurs when multiple live migrations are performed simultaneously. In all these cases, the available network bandwidth for a live migration dynamically changes.
- When several VMs are co-located, they compete with each other for obtaining CPU resources. This contention may lead to performance degradation of workloads. Hence, the memory update speed of the VM dynamically changes due to the activities of other co-located VMs.

A naive model that miscalculates migration times and possibly under- or over-estimates the corresponding network traffic will result in erroneous simulation results far from the real world behaviors. To accurately estimate the impact of the memory update speed, the resource sharing amongst workloads and VMs must be considered. In other words, the simulation framework should consider contention on virtualized and non-virtualized systems: If VMs are co-located on a physical machine, the system should compute a correct CPU time to each VM. If a network link is shared with multiple data transfers, including those of migrations, the system needs to assign a correct bandwidth to each transfer.

3.2.3 *Implementation*

The live migration model we implemented in SimGrid, performs the same operations of the precopy algorithm described in Sec. 3.2.1. We denote the memory size of a VM as R. The memory update intensity of the VM is given by a parameter α (bytes/flops). It denotes how much memory pages are marked dirty while one CPU operation (in a simulation world) is executed. We have published a Qemu extension (Qemu-DPT, dirty page tracking) to easily measure the current memory update speed of a VM, which enables SimGrid users to easily determine the memory update intensity of the workloads they want to study. For example, a user launches a VM with Qemu-DPT, starts a workload on the VM, changes a request rate to the workload, and measures the current CPU utilization level and the memory update speed. Users can easily confirm whether there is a linear correlation and determine the value of correlation coefficient (i.e., the memory update intensity of the workload). If a linear correlation is not appropriate, it is possible to develop another correlation from measured data.

During a live migration, our model repeats data transfer until the end of Stage 3. All transfers are simulated by using the `send/recv` operations proposed by the SimGrid MSG API. We define the data size of the ith transfer as X_i bytes. At Stage 1, the precopy algorithm sends all the memory pages, *i.e.*, $X_1 = R$. In Stage 2, the algorithm sends memory pages updated during the previous data transfer. Based on our preliminary experiments, we assume that there will be many situations where the memory update speed is roughly proportional to the CPU utilization level. We use a linear correlation function as the first example of correlation functions. The size of updated memory pages during the data transfer is proportional to the total amount of CPU shares assigned to the VM during this period. Thus, we obtain the data transfer size of the next phase as $X_{i+1} = \min(\alpha S_i, R')$, where S_i (floating operations, flops) is the total amount of CPU shares assigned to the VM during the ith data transfer (as a reminder, S_i is computed by the solver engine as described in Sec. 3.1). R' is the memory size of the working set used by workloads on the VM. The size of updated memory pages never exceeds the working set size.

The simulation framework formulates constraint problems to solve the duration of each data transfer, which is based on the network speed, latency, and other communication tasks sharing the same network link. If the maximum throughput of migration traffic B is given, the model controls the transfer speed of migration data, not to exceed this throughput. This parameter corresponds to `migrate-set-speed` of the libvirt API [8].

Every time migration data is sent in a simulation world, the model estimates the available bandwidth for the ongoing live migration, in the same manner as the hypervisor does in the real world. The current throughput of migration traffic is estimated by dividing the size of sent data by the time required for sending the data. This estimated value is used to determine the acceptable size of remaining data when migrating from Stage 2 to Stage 3. If the maximum downtime d is 30 ms (*i.e.*, the default value of Qemu), and if the migration bandwidth is estimated at 1 Gbps, the remaining data size must be less than 3,750 KB in Stage 3. The migration mechanism repeats the iteration of Stage 2 until this condition is met.

Finally, at Stage 3, our model creates the communication task of X_n bytes to transfer the rest of memory pages. After this final task ends, the system switches the execution host of the VM to the destination.

Since a linear correlation will explain the memory update speed in typical situations, we implemented it as a default correlation function. However, the proportional function used in the current implementation is just an example of correlation functions to estimate the memory update speed of a VM. If this proportional function is not appropriate for a situation, it is possible to define another correlation function.

The correlation function of the memory update speed can be defined not only with the CPU utilization level but also with any other variables available in the simulation world. For example, a request rate of an application can be used, if it explains the memory update speed of a VM. We consider that in most cases a user will need to modify only one function `get_update_size()` in the VM extension, which calculates the size of updated memory pages.

3.3 SimGrid VM API (C and Java)

In our work, we extended the MSG programming API in order to manipulate a VM resource object as shown in Table 1. Each operation in this API corresponds to a real-world VM operation such as create/destroy, start/shutdown, resume/suspend and migrate. We newly defined the `msg_vm_t` structure, which is a VM object in a simulation world, supporting these VM APIs. It should be noted that a VM object inherits all features from a PM object `msg_host_t`. A VM resource object supports most existing operations of a PM, such as task creation and execution. From the viewpoint of users, they can treat a VM as an ordinary host, except a VM supports these VM-specific operations.

As discussed in Sec. 3.2.3, users need to specify the memory size of a VM, the memory update intensity of the VM, and the memory size of the working set of memory used by a workload on the VM. These parameters are specified either at the VM creation or through the `MSG_VM_set_params()` function.

Fig. 5 shows an example code using the VM APIs. `example()` starts a VM on the given PM, and launches a worker process on the VM. The way of launching a process is exactly the same as that of a PM; we can use `MSG_process_create()` also for a VM.

Although this example is in `C`, it is noteworthy that the JAVA SimGrid API has been also extended. Hence, end-users can develop their simulators either by interacting with the native C routines or by using the JAVA bindings.

Finally, we highlight that we also extended the multicore support of SimGrid to allow the simulation of virtualized systems running on multicore servers. The `set_affinity` function pins the execution of a VM (or a task) on given CPU cores.

4 Evaluation of VM Extensions

In order to confirm the correctness and the accuracy of the VM extensions within SimGrid, we conducted several micro benchmark experiments on Grid'5000 and compared results with the simulated ones. We discuss in this section major ones.

TABLE 1: The APIs to manipulate a VM resource object in the VM support

`msg_vm_t MSG_VM_create(msg_host_t pm, ...)`	Create a VM object on the given PM with the specified parameters.
`void MSG_VM_destroy(msg_vm_t vm)`	Destroy the VM.
`void MSG_VM_start(msg_vm_t vm)`	Start the VM.
`void MSG_VM_shutdown (msg_vm_t vm)`	Shutdown the VM.
`void MSG_VM_migrate(msg_vm_t vm, msg_host_t dst_pm)`	Migrate the VM to the given destination PM.
`void MSG_VM_set_params(msg_vm_t vm, ws_params_t params)`	Set parameters of the VM.
`vm_state_t MSG_VM_get_state(msg_vm_t vm)`	Return the state of the VM.
`msg_host_t void MSG_VM_get_pm(msg_vm_t vm)`	Return the PM of the VM.
`void MSG_VM_suspend(msg_vm_t vm)`	Suspend the execution of the VM. Keep VM states on memory.
`void MSG_VM_resume(msg_vm_t vm)`	Resume the execution of the VM.
`void MSG_VM_save(msg_vm_t vm)`	Suspend the execution of the VM. Save VM states to storage.
`void MSG_VM_restore(msg_vm_t vm)`	Restore the execution of the VM from storage.
`void MSG_VM_set_bound(msg_vm_t vm, double bound)`	Set the maximum CPU utilization level of the VM.
`void MSG_VM_set_affinity(msg_vm_t vm, unsigned long mask)`	Set the CPU-core affinity of the VM.

```c
/* the main function of the worker process */
static int worker_main(int argc, char **argv)
{
        /* computation size (floating operations) */
        const double flops = 10000;

        /* Repeat computation. */
        for (;;) {
                msg_task_t task = MSG_task_create("Task", flops, 0,
                        NULL);
                MSG_task_execute(task);
                MSG_task_destroy(task);
        }
        return 0;
}

void example(msg_host_t pm)
{
        unsigned long ramsize = 2UL * 1024 * 1024; // 2 GB
        double memory_update_intensity = 60; // 60 MB/s at 100 % CPU
            load
        double working_set_size = 0.9; // 90 % of ramsize

        /* 0. Create a VM (named VM0) on the PM. */
        msg_vm_t vm = MSG_vm_create(pm, "VM0", ramsize,
            mem_update_intensity, working_set_size);
        MSG_vm_start(vm);

        /* 1. Launch a process on the VM. */
        msg_process_t pr = MSG_process_create("worker", worker_main,
            NULL, vm);

        /* 2. Keep the VM running for 10 seconds. */
        MSG_process_sleep(10);

        /* 3. Suspend the VM for 10 seconds. */
        MSG_vm_suspend(vm);
        MSG_process_sleep(10);
        MSG_vm_resume(vm);

        /* 4. Keep the VM running for 10 seconds. */
        MSG_process_sleep(10);

        /* 5. Clean up. */
        MSG_process_kill(pr);
        MSG_vm_shutdown(vm);
        MSG_vm_destroy(vm);
}
```

Fig. 5: An Example Code Using the VM APIs

4.1 Experimental Conditions

In this section, we give details that will enable to reproduce the experiments discussed in the next sections.

All experiments in the real world were conducted on the Grid'5000 Graphene cluster. Each PM has one Intel Xeon X3440 (4 CPU cores), 16 GB memory, and a GbE NIC. The hardware virtualization mechanism (*i.e.*, Intel VT) was enabled.

We used Qemu/KVM (Qemu-1.5 and Linux-3.2) for the hypervisor in the experiments. Assuming long-lived active VMs instead of idle VMs that never became active after being booted, we modified a few lines of source code of Qemu to disable the mechanism not to transfer zero-filled pages. This mechanism does not effectively work if the VM is running for a while with active workloads. In such case, non-zero data already exists in most memory pages. Moreover, Qemu-1.5 also supports the XBRLE compression of migration data [12]. This mechanism, which is disabled in the default settings of major Linux distributions, enables to pick up updated regions of the pages and send them with compression (even though a memory page is marked as dirty, only a few bytes in the page may have been updated, thus selecting only the updated data enables to reduce the migration traffic). Although, it is possible to extend SimGrid to simulate the behaviors of these compression mechanisms, we choose to focus our study on the development of a sound model that can capture common behaviors among hypervisors, and to not focus on implementation-specific details of a particular hypervisor. Hence, we kept this mechanism disabled. The virtual disk of a migrating VM is shared between source and destination physical machines by a NFS server. This is a widely-used storage configuration in cloud computing platforms. To cover more advanced configurations, we are extending our model to support virtual disks and understand the impact of I/O intensive workloads. This effort enables us to simulate the relocation of the associated VM images, which will be reported in our future work.

We used the Execo automatic deployment engine [13] to describe and perform the experiments by using its Python API. According to the scenario of an experiment, the Execo deployment engine automatically reserves, install and configure nodes and network resources that are mandatory before invoking the scripts of the experiment. This mechanism allows us to easily run and reproduce our experiments. All experiments were repeated at least 5 times with no noticeable deviations in obtained results. Then, the same experiments were also conducted on SimGrid.

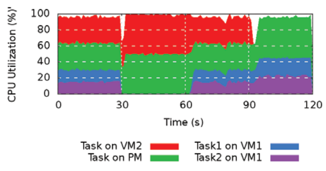
Although the VM extension of SimGrid supports multicore CPU simulation, in the following micro benchmarks, VMs were pinned to the first CPU core both in real-world and simulation experiments, so as to carefully discuss how resource contention impacts on live migration performance.

Finally, in order to carefully investigate live migration behaviors, we developed a memory update program, `memtouch`, emulating various load conditions by real applications. The `memtouch` program works as a workload that has a linear correlation between CPU utilization levels and memory update speeds. It produces the memory update speed at a given CPU utilization level, by interleaving busy loops, memory updates and micro sleeps in an appropriate ratio as explained in our previous work [14].

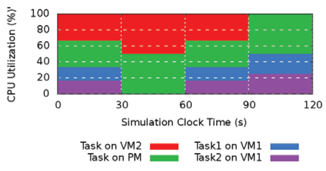
The memory update program accepts two kinds of parameters. One is a target CPU utilization level (%). The other is a memory update intensity value that characterizes an application. For example, we observed that the database workload in Sec. 3.2.2 had a linear correlation between memory update speeds (Y) and CPU utilization levels (X), which was $Y = \alpha X$. This α is the key parameter to model the memory update behavior of this database workload. From Fig. 3c, we can roughly estimate the memory update intensity α to be *60MB/s at CPU 100%*. This 60MB/s is passed to the arguments of the memory update program. If a given target CPU utilization level is 50%, the memory update speed of the program becomes 30MB/s. Moreover, if other workloads or co-located VMs compete for CPU resource and the memory update program only gets 25%, the actual memory update speed becomes 15MB/s. This behavior correctly emulates what happens in consolidated situations.

4.2 Evaluation of the VM Workstation Model

The CPU resource allocation of a VM impacts on is memory update speed, and the memory update speed is a dominant parameter that governs live migration time. Before doing the experiments focusing on live migration, we confirmed that our VM support of SimGrid correctly calculates CPU and network resource allocations for VMs.



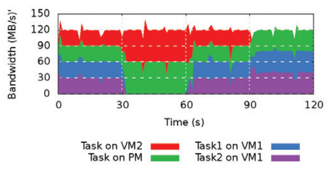
(a) Grid'5000



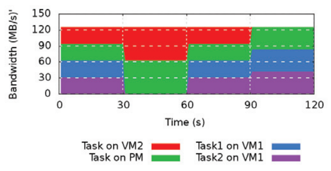
(b) Simulation

Fig. 6: The CPU load of each task in a CPU resource contention experiment

In each graph, from the top to the bottom, the CPU usages of *Task on VM2*, *Task on PM*, *Task1 on VM1*, and *Task2 on VM1*, are illustrated, respectively.



(a) Grid'5000



(b) Simulation

Fig. 7: The network throughput of each task in a network resource contention experiment

In each graph, from the top to the bottom, the network throughput of *Task on VM2*, *Task on PM*, *Task1 on VM1*, and *Task2 on VM1*, are illustrated, respectively.

We launched 2 VMs and 4 computation tasks with the same arrangement as the right side of Fig. 2: 2 VMs (VM1 and VM2) are launched on a PM. VM1 has 2 computation tasks, and VM2 has 1 computation task. The PM also has a computation task. All tasks tried to obtain the 100% CPU utilization, competing with each other. We used the memory update intensity of the database benchmark as discussed in the above (*i.e.*, 60MB/s at the 100% CPU utilization).

Fig. 6a shows the CPU utilization level of each task in a real-world experiment on Grid'5000. First, VM1, VM2, and the task on the PM fairly shared the CPU resource of the PM, consuming approximately 33.3% respectively. On VM1, each task consumed approximately 50% of the CPU share assigned to VM1. At 30 seconds, we suspended VM1 running the 2 tasks. Thus, the task on VM2 and the task on the PM consumed approximately 50%, respectively. At 60 seconds, we resumed VM1. The CPU utilization of each task was recovered as the initial state. At 90 seconds, we shut down VM2. Then, the CPU share of VM1 and the task on the PM increased to approximately 50%, respectively. These results were reasonable, considering the hypervisor fairly assigns CPU resources to each VM and the task on the PM.

The same experiment was performed in simulation. As shown in Fig. 6b, the VM support of SimGrid correctly captured the CPU load change of each task in large part. However, there are minor differences between the real-world and simulation experiments, especially just after a VM operation (i.e., suspend/resume and shutdown) was invoked. For example, the shutdown of VM2 took approximately 3 seconds to be completed. When the suspension of a VM is invoked, the guest OS stops all the processes on it, and then commits all pending write operations to virtual disks. In the real world, these operations will sometimes have an impact on other tasks on the same PM. We consider that if a user needs to simulate further detail of VM behaviors, it is possible to add the overhead of VM operations into the VM model. As mentioned earlier, we are working for example on modeling VM boot and snapshotting costs related to I/O accesses to the VM images.

The second experiment we conducted aimed at observing the behavior of VMs under network resource contention. Because live migration time is impacted also by network resource availability, the VM support of SimGrid needs to correctly calculate network resource assignments to each communication task. Although a former study regarding SimGrid [1] already proved the correctness of its network model, the study had been done when our VM support was not available. It is necessary to confirm the network model also correctly works at the VM level.

We launched 2 VMs and 4 communication tasks with the same arrangement as the previous experiment. However, in this experiment, 4 communication tasks were launched instead of computation tasks. Each communication task continued to send data to another PM. The destination PM of each communication task was different, respectively.

Fig. 7a shows the result of a real-world experiment. `iperf` was used to send data to destination. The host operating system fairly assigned network resources to each TCP connection. For example, while VM1 was suspended, the 2 communication tasks on it could not send data, and the bandwidth of the 1Gbps link was split into 2 other TCP connections. As shown in Fig. 7b, the VM support of SimGrid correctly captured this behavior.

It should be noted that, in the real world experiment, the packet queuing discipline of the physical network interface of the source PM needed to be set to SFQ (Stochastic Fairness Queuing), which enforces bandwidth fairness among TCP connections more strongly than the default one of Linux (i.e., basically, first in first out). Although the TCP algorithm is designed to achieve fairness among connections, we experienced that, without using SFQ, the network traffic made by the task running on the PM occupied more than 90% of the available bandwidth of the network interface. We consider that this problem was caused by the difference of packet flow paths between the task on the PM and the other tasks running on VMs. Although the latest libvirt code on its development repository uses SFQ as we did, libvirt-0.9.12[2] used in the experiments did not.

In summary, through these experiments, we confirmed that the VM support of SimGrid correctly calculates CPU and network resource assignments to VMs and tasks. The prerequisite to obtain sound simulation results of live migrations is met.

4.3 Live migrations with various CPU levels

We conducted a series of experiments to confirm the correctness of the live migration model. The settings

2. It is included the latest stable release (wheezy) of the Debian/GNU distribution.

of VMs were the same as the above experiments. The memory update intensity was set to that of the database benchmark (*i.e.*, 60MB/s at the 100% CPU utilization). The memory size of a VM was 2GB. The size of the working set memory was set to its 90%.

Fig. 8 shows live migration times with different CPU utilization levels. This graph compares the real experiments on Grid'5000, the simulation experiments using our migration model, and the simulation experiments with the naive migration model (*i.e.*, without considering memory updates). These results confirm the preliminary investigations performed in Sec. 3.2.1: As the CPU utilization level of the VM increased, we observed that the live migration time of the VM became longer. Our simulation framework implementing the precopy algorithm successfully simulated this upward curve, within 2 seconds deviations (9% at most).

On the other hand, the naive simulation without the precopy algorithm failed to calculate correct migration times, especially in the higher CPU utilization levels. At the CPU 100% case, the naive simulation underestimated the migration time by 50%. Simulation frameworks without the migration model, e.g., CloudSim, will produce these results.

The small differences between the Grid'5000 results and our precopy model, *e.g.*, up to 2 seconds in Fig. 8, can be explained by the fact that in addition to the `memtouch` program, other programs and the guest kernel itself are consuming CPU cycles and are slightly updating memory pages of the VM in the reality. Furthermore, because the system clock on the guest OS is less accurate than that of the host OS, the memory update speed by the program involves small deviations from the target speed. We are going to show that this deviation might be problematic in a few corner cases where memory update speed and available migration bandwidth are close to each other.

4.4 Live Migrations under CPU Contention

A live migration is deeply impacted by CPU resource contention on the source and destination PMs. We performed migration experiments with the different numbers of co-located VMs. In addition to the VM to be migrated, other VMs were launched on the source or destination PM. To discuss serious resource contention, sometimes found dynamic VM packing systems, all the VMs were pinned to the first physical CPU core of the PM. The `cpuset` feature of libvirt [8] was used in real-world experiments, and `MSG_vm_set_affinity()` was called in simulation programs. All the VMs executed the `memtouch` program on their guest OSes. The target CPU utilization of `memtouch` was set to 100%; although all the VM on the PM tried to obtain 100% CPU resource, they actually obtained partial CPU resource due to co-location. Qemu/KVM will fairly assign CPU resource to each VM. [3]

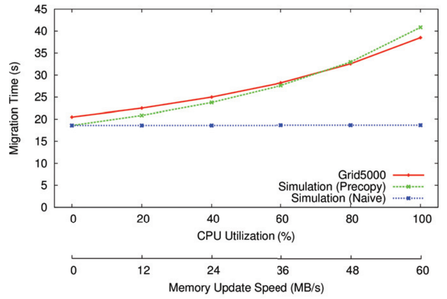


Fig. 8: Comparison of live migration time

The X axis shows CPU utilization levels and corresponding memory update speeds. Update memory speed of the workload is 60 MB/s at 100% of CPU.

Fig. 9a shows live migration times when other VMs were launched on the source PM. When the number of co-located VMs was higher, the actual live migration times decreased, becoming close to that of no memory update case (*i.e.*, approximately 20 seconds). This serious CPU resource contention reduced the actual memory update speed of the migrating VM, which results in shorter migration times. Our simulation framework correctly calculated assigned CPU resource and captured migration behavior.

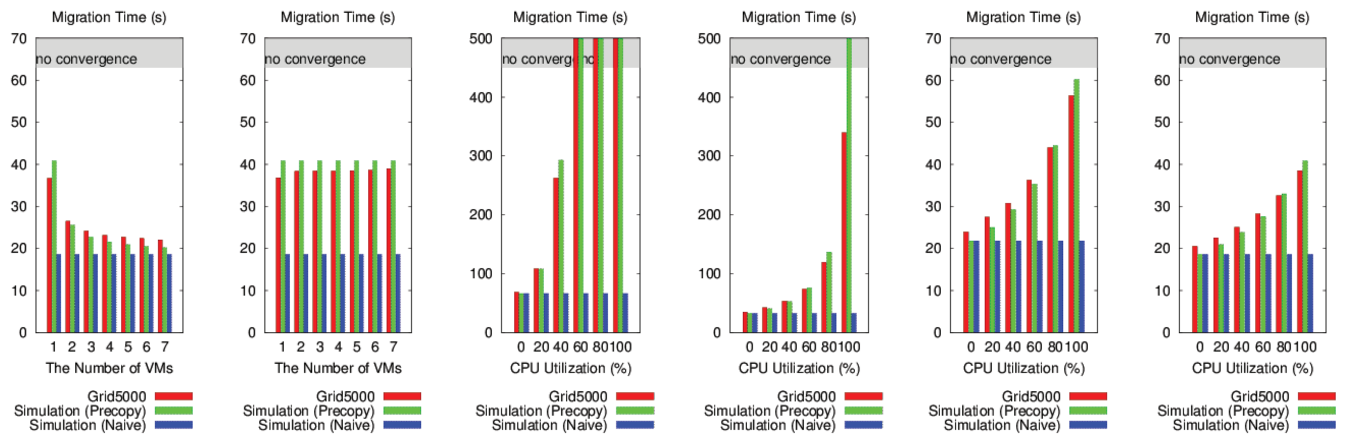
Fig. 9b is the case where the other VMs were launched on the destination PM. Because the migrating VM, updating memory pages on the source PM, did not compete with the other VMs for CPU resource, the live migration times were not affected by the number of other VMs. This behavior was successfully reproduced by our simulation framework with the precopy model.

When there were other VMs on the destination PM, the migration times on Grid'5000 were slightly longer than that of the 1-VM case. During a live migration, the hypervisor launches a dummy VM process on the destination PM. The dummy VM process is receiving migration data from the source PM. Although this receiving operation requires very little CPU resource, the data receiving speed was slightly reduced due to the CPU resource contention on the destination PM. We consider that CPU overheads of sending/receiving migration data are negligible in most use-cases because they are substantially small as compared to resource usage by VM workloads. However, if a user of our simulation framework needs to carefully simulate such behaviors, it is possible to create micro computation tasks corresponding to data sending/receiving overheads by leveraging the SimGrid MSG API. In addition, as discussed in Sec. 4.3, there were also small deviations between the results of Grid'5000 and the simulation. If these deviations are not negligible, it is also possible to improve the simulation mechanism by taking into account CPU cycle consumption and memory update by the guest kernel and other processes.

4.5 Live Migrations under Network Contention

A live migration time is also deeply impacted by the available network bandwidth for the migration. The hypervisor uses a TCP connection to transfer migration data of a VM. It should be noted that from the viewpoint

3. It should be noted that even if multiple VMs share one PM, the number of memory pages associated to each VM does not change. The hypervisor assigns 2 GB of memory pages to each VM.



(a) SRC (b) DST (c) Bw. 25% (d) Bw. 50% (e) Bw. 75% (f) Bw. 100%

Fig. 9: Live migration time with resource contention

In Figs. 9a and 9b, the number of co-located VMs is changed, on the source and destination PM, respectively. In Figs. 9c-9f, the available bandwidth is limited, *i.e.*, 25%, 50%, 75%, and 100% of GbE, respectively. The results on the gray zone (no convergence) mean the cases where migrations never finished. Note that the scale of the Y axis is different. Update memory speed of the workload is 60 MB/s at 100% of CPU.

of the host OS this is a normal TCP connection opened by a userland process (i.e., the Qemu process of a VM). On the host OS and the network link, there is no difference between the TCP connection of a live migration and other TCP connections (including NFS sessions used for disk I/O of VMs). We limited the available bandwidth for a migration to 25%, 50%, 75% and 100% of the GbE throughput. The virsh `migrate-set-speed` command was used. These experiments are intended to correspond to real-world situations where concurrent live migrations are performed at once or live migration traffic is affected by other background traffic. Because disk I/O of a VM creates NFS requests, these experiments also cover situations where disk I/O of VMs impact live migration traffic. Fig. 9c-9f compared real and simulated migration times. When the available bandwidth was smaller, the simulation with the naive model underestimated live migration times more seriously. The real migration times exponentially increased, as the available bandwidth was smaller. The precopy live migration model correctly simulated these trends in most cases. If the available bandwidth is smaller than the actual memory update speed of the VM, the algorithm of the precopy live migration never finished. The iterative memory copy phase of the precopy live migration (i.e., Stage 2 in Sec. 3.2.1) continues until someone cancels (i.e., gives up) the live migration. This behavior was correctly simulated at the 60%, 80% and 100% CPU utilization levels of the 25% GbE bandwidth case (Fig. 9c).

The only exceptional case where the precopy model did not follow the results of real experiments, was at the 100% CPU utilization level of the 50% GbE bandwidth case (Fig. 9d). The simulation using the precopy model predicted this live migration never finished, although the live migration on Grid'5000 finished in 330 seconds. In this condition, the migration bandwidth was 50% of 1 Gbps (*i.e.*, 59.6MB/s), while the memory update speed was 60MB/s. Because the migration bandwidth is theoretically smaller than the memory update speed, the precopy migration model iterated Stage 2 of the precopy algorithm forever. On the other hand, as discussed in Sec. 4.3, since the actual memory update speeds of the VM would be slightly below the migration bandwidth, the real live migration finished in a finite period of time.

We consider that this problem will not appear in most cases. However, if a user needs to simulate such an exceptional situation where the migration bandwidth and the memory update speed of a VM are very close, it is necessary to give careful consideration to the accuracy of these simulation parameters. A simulation program may need to consider subtle memory updates by the guest kernel and the network bandwidth fluctuation caused by other background traffic.

To conclude, we can affirm that our experiments were accurate enough to validate our extensions in most cases.

5 A First Use Case

To strengthen the confidence of the accuracy of our extensions within the SimGrid toolkit and also to illustrate their usefulness, we implemented a first program dealing with the well-known VM placement problem. VM placement problem is a combinational optimization determining which VMs should be placed on which PMs. In this section, our VM placement mechanism dynamically migrates VMs among PMs in order to resolve the overloading of PMs.

We believe that such a use-case is relevant as it will enable researchers to investigate new VM placement policies without performing large-scale in-vivo experiments.

5.1 Overview

The framework is composed of two processes. The first one creates n VMs, each of which is based on one of predefined VM classes. A VM class is a template of the specification of a VM and its workload. It is described as `nb_cpu:ramsize:net_bw:mig_speed:mem_speed`. VMs are launched on PMs in a round-robin manner, *i.e.*, each PM has almost the same number of VMs. Next, the process repeatedly changes target CPU loads of VMs.

Every t second, it selects one VM and changes its CPU load according to a Gaussian distribution. t is a random variable that follows an exponential distribution with rate parameter λ. The Gaussian distribution is defined by a particular mean (μ) as well as a particular standard deviation (σ) that are given at each simulation.

The second process controls a relocation the VMs each time it is needed. Every 60 seconds, the process checks CPU loads of VMs, and if it detects an overloading PM (*i.e.*, a PM that cannot satisfy the VMs expectations), it invokes the Entropy algorithm ([15], [16]) to solve the VM placement problem. Roughly speaking, the Entropy process is composed of two phases. During the first one, *Computation phase*, Entropy looks for a viable placement by solving a constraint satisfaction problem, then it calculates an optimal reconfiguration plan (i.e. the order of migrations that should be performed to switch from the previous reconfiguration to the new ones). The second phase consists in applying the returned reconfiguration plan. It is noteworthy that because VMPP is an NP-hard problem, the duration of the computation phase is time-bound with a predefined value set to $min(nb\_nodes/8; 300)$.

5.2 Experimental Conditions and Results

The program has been implemented in Java on top of SimGrid and in the reality[4]. A specific seed for the random-based operations enabled us to ensure reproducibility between the different in-vivo executions and the simulations. The *in vivo* experiments have been performed on top of the Grid'5000 Graphene cluster with the same conditions (Linux 3.2, Qemu 1.5 and SFQ network policy enabled, see 4.1).

In order to reach over-provisioning situations, 6 VMs per node are initially launched. Each VM has been created as one of the 8 VM classes. Each VM class here corresponded to a type of workload with different memory update intensity, *i.e.*, 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 previously described. The parameters were $\lambda = Nb\_VMs/300$ and $\mu = 60$, $\sigma = 20$. Concretely, the load of each VM varied on average every 5 min in steps of 10 (with a significant part between 40% and 80%). The `memtouch` program introduced in Sec. 4.1 has been used to stress both the CPU and the memory accordingly. The duration of the experiment was set to 3600 seconds.

The simulated environment reflects the real 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).

4. Codes are available on the github beyond the cloud pages, respectively at the VMPlaceS and VMPlaceS-G5K repositories (http://beyondtheclouds.github.io).

Fig. 10 shows the cost of the two phases of the Entropy algorithm for each invocation when considering 32 PMs and 192 VMs through simulations (top) and in reality (bottom). At coarse-grained, we can see that simulation results successfully followed in-vivo results. Diving into details, the difference between the *in-siclo* 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 consider that the current simulation results are sufficiently accurate to capture performance trends. If necessary, we can furthermore analysis the influence of this fluctuation by adding some bandwidth deviations to competing TCP sessions in the simulation world. Finally, we found that, amongst the 170 GB that have been transferred during the complete experiment, a bit more than 10% was caused by the data transfer of Stage 2 and Stage 3. The investigated algorithm performed migrations when PMs were overbooked (i.e, serious resource contention happened, and the memory intensity was less impacting on migration times). Our virtualization mechanism successfully simulated these behaviors by taking into account resource contentions amongst VMs.

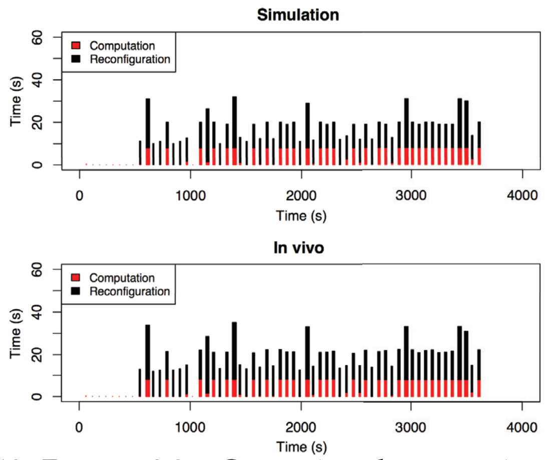


Fig. 10: Entropy 2.0 - Comparison between simulated and in-vivo observations

The red parts correspond to the time periods where Entropy checks the viability of the current configuration and compute a new viable configuration if necessary. The black parts correspond to the application phase of reconfiguration plans (*i.e.*, the migrations of the VMs are performed to apply a new reconfiguration plan).

To illustrate the relevance of the simulator framework, we conducted the same experiments with 2048 PMs/12288 VMs. The experiment ran in approximately two hours. Table 2 presents the results of the five invocations of Entropy that has been performed during the 3600 simulated seconds. We also conducted the same experiments with 4096 PMs and 24576 VMs. This simulation lasted approximately 8 hours. Although the simulation time exponentially increased as the number of nodes increased, it is possible to simulate a large-scale, complex scenario comprising 10 thousands of nodes just in one day. We are now optimizing simulation code for further performance improvement.

Simulation clock time when reconfig. was invoked (sec)	**60**	**120**	**433**	**796**	**1597**
Computation duration (sec)	9	301	301	301	301
Reconfiguration duration (sec)	not needed	12	62	500	not finalized
Nb of migrations	0	2	74	227	483

TABLE 2: The cost of each reconfiguration during the 2048 PMs/12288 VMs simulation

Due to the time to perform the computation and reconfiguration phases, Entropy was invoked only five times during the simulation.

The experiments enabled us to confirm that Entropy2.0 is not scalable as it did not succeed to find a viable solution within the 300 seconds timeout and hence did not perform any reconfiguration. Discussing in details the efficiency of the Entropy algorithm is out of the scope of the paper. However, thanks to the simulator we can see some important phenomenons. In the first hundreds of seconds, the cluster did not experience replacement of VMs, because the loads of VMs were still small. However, once Entropy detected non viable configuration, applying a reconfiguration plan was much more time-consuming than computing it. The longest reconfiguration plan did not complete in more than 1702 seconds. This result means that VM placement problem also needs to address the way of shorten reconfiguration phases, not only that of computing phases. Leveraging the SimGrid toolkit enables us to observe detailed system behaviors without facing with the burden of conducting large scale experiments.

6 Related Work

CloudSim [3] is a simulation framework that allows users to develop simulation programs of cloud datacenters. It has been used, for example, for studies regarding dynamic consolidation and resource provisioning. Although it looks that CloudSim shares the same goal with our SimGrid project, the current API of CloudSim is based on a relatively top-down viewpoint of cloud environments. Their API provides a perspective of datacenters composed of application services and virtual and physical hosts. Users will create pseudo datacenter objects in their simulation programs. In their publication, a migration time is calculated by dividing a VM memory size by a network bandwidth. [5] This model, however, cannot correctly simulate many real environments where workloads perform substantial memory writes. On the other hand, we can say that SimGrid takes a bottom-up approach. Our on-going project currently pays great attention to carefully simulate the behavior of a VM running in various conditions, which leads to well-fitting simulation results of cloud environments where many VMs are concurrently running with various workloads. As far as we know, our virtualization support of SimGrid is the first simulation framework that implements a live migration model of the precopy algorithm. It will provide sound simulation results for dynamic consolidation studies. The SONGS project, supporting our activity to add virtualization abstractions into SimGrid, is also working on providing the modeling of datacenters. The same level of the API of Amazon EC2 is supported in the ongoing work.

5. In the source code of the latest release of CloudSim (i.e., cloudsim-3.0.3), the power-aware datacenter model of CloudSim (PowerDataceter Class) assumes that the half network bandwidth of a target host is always used for a migration because they assume the other half is used for other VM communications. According to FAQ of CloudSim [17], for those who directly program migrations in simulations, CloudSim provides an API to invoke a migration. However, users need to assign an estimated completion time to each migration. It does not have a mechanism to accurately estimate the completion time.

iCanCloud [18] is a simulation framework with the job dispatch model of a virtualized datacenter. Their hypervisor module is composed of a job scheduler, waiting/running/finished job queues, and a set of VMs. For the use of commercial cloud services like Amazon EC2, there is trade-off between financial cost and application performance. This framework was used to simulate provisioning and scheduling algorithms with the cost-per-performance metric. Koala [19] is a discrete-event simulator emulating the Amazon EC2 interface. It extends the cloud management framework Eucalyptus [20], modeling its cloud/cluster/node controllers. VM placement algorithms were compared using this simulator. These frameworks are designed to study a higher-level perspective of cloud datacenters, such as resource provisioning, scheduling and energy saving. Contrary, SimGrid, originally designed for the simulation of distributed systems, performs computation and data sending/receiving in the simulation world. It simulates more fine-grained system behavior, which is necessary to thoroughly analyze cloud environments.

GreenCloud [21] is a simulator extending a network simulator NS2. It allows simulating energy consumption of a datacenter, considering workload distributions and network topology. This simulator is intended to capture communication details with their packet-level simulation. SimGrid does not simulate distributed system in the packet level, but in the communication flow basis. It is designed to simulate distributed systems, composed of computation and communication, in a scalable manner. The SONGS project is also working on integrate energy models to SimGrid, which enables energy consumption simulations of datacenters.

We consider that it would be possible to implement a precopy live migration model on these simulation toolkits. However, substantial extension of existing code may be necessary. In order to correctly simulate migration behaviors of VMs, a simulation toolkit requires the mechanism modeling a live migration algorithm and the mechanism to correctly calculate resource share of VMs.

SimGrid supports a formal verification mechanism for distributed systems, a simulation mechanism of parallel task scheduling with DAG (direct acyclic graphs) models, and a simulation mechanism of unmodified MPI applications. The VM extension of SimGrid enables researchers to use these mechanisms also for virtualized environments. We have carefully designed the VM extension to be compatible with existing components in SimGrid.

7 Conclusion

We are developing a scalable, versatile, and easy-to-use simulation framework supporting virtualized distributed environments, which is based on a widely-used, open-source simulation framework, SimGrid. The extension to SimGrid is seamlessly integrated with existing components of the simulation framework. Users can easily develop simulation programs comprising VMs and PMs through the same SimGrid API. We redesigned the constraint problem solver of SimGrid to support resource share calculation of VMs, and developed a live migration model implementing the precopy migration algorithm of Qemu/KVM. Through micro benchmarks, although we observed that a few corner cases cannot be easily simulated when the memory update speed is closed to the network bandwidth, we confirmed that our precopy live migration model reproduced sound simulations results in most cases. In addition, we showed that a naive migration model, not considering memory updates of the migrating VM nor resource sharing competition, underestimated the live migration time as well as the resulting network traffic. We illustrated the advantage of our simulation framework by implementing and discussing a first use-case dealing with dynamic placement of VMs. Although we succeeded to perform simulations up to 4K PMs and 25K VMs, we discovered that the scalability of our extensions is exponential and not proportional to the number of PMs/VMs. We are working with the SimGrid core developers to investigate how the performances of our extensions can be improved. The first envisioned approach is to use co-routines instead of the `pthread` library. Finally, we highlight that our extensions have been integrated into the core of SimGrid since its version 3.11, released in May 2014.

References

[1] H. Casanova, A. Legrand, and M. Quinson, "Simgrid: a generic framework for large-scale distributed experiments," in *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, 2008, pp. 126–131.

[2] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, sep 2009, pp. 99–100.

[3] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[4] "Simgrid publications," http://simgrid.gforge.inria.fr/Publications.html.

[5] L. M. Schnorr, A. Legrand, and J.-M. Vincent, "Detection and analysis of resource usage anomalies in large distributed systems through multi-scale visualization," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 15, pp. 1792–1816, 2012. [Online]. Available: http://dx.doi.org/10.1002/cpe.1885

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005, pp. 273–286.

[7] A. Kivity, "kvm: the Linux virtual machine monitor," in *OLS '07: The 2007 Ottawa Linux Symposium*, Jul. 2007, pp. 225–230.

[8] "libvirt: The virtualization api," http://libvirt.org.

[9] J. Fulmer, "Siege," http://www.joedog.org/siege-home.

[10] The PostgreSQL Global Development Group, "Postgresql," http://www.postgresql.org/.

[11] Transaction Processing Performance Council, "TPC benchmark b, standard specification revision 2.0," http://www.tpc.org/tpcb/spec/tpcb_current.pdf, Jun 1994.

[12] P. Svard, J. Tordsson, B. Hudzia, and E. Elmroth, "High performance live migration through dynamic page transfer reordering and compression," in *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011, pp. 542–548.

[13] INRIA, "Execo:a generic toolbox for conducting and controlling large-scale experiments," http://execo.gforge.inria.fr/.

[14] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Reactive cloud: Consolidating virtual machines with postcopy live migration," *IPSJ Transactions on Advanced Computing Systems*, vol. ACS37, pp. 86–98, 2012.

[15] F. Hermenier, X. Lorca, J. M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Mar. 2009, pp. 41–50.

[16] F. Hermenier, J. Lawall, and G. Muller, "Btrplace: A flexible consolidation manager for highly available applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 99, p. 1, 2013.

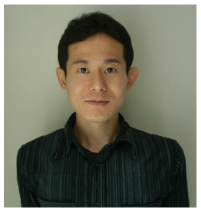
[17] CloudSim FAQ (retrieved August 13th 2015), "Advanced features: 1. How can I code VM migration inside a data center?" https://code.google.com/p/cloudsim/wiki/FAQ.

[18] A. Nunez, J. Vazquez-Poletti, A. Caminero, G. Castane, J. Carretero, and I. Llorente, "icancloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, 2012.

[19] K. Mills, J. Filliben, and C. Dabrowski, "Comparing vm-placement algorithms for on-demand clouds," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE Computer Society, 2011, pp. 91–98.

[20] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009, pp. 124–131.

[21] D. Kliazovich, P. Bouvry, Y. Audzevich, and S. Khan, "Greencloud: A packet-level simulator of energy-aware cloud computing data centers," in *Global Telecommunications Conference (GLOBECOM 2010)*, 2010, pp. 1–5.



Takahiro Hirofuchi Dr. Takahiro Hirofuchi is a senior researcher of National Institute of Advanced Industrial Science and Technology (AIST) in Japan. He is working on virtualization technologies for advanced cloud computing. He obtained a Ph.D. of engineering in March 2007 at Nara Institute of Science and Technology (NAIST). He obtained the BS of Geophysics at Faculty of Science in Kyoto University in March 2002. He is an expert of operating system, virtual machine, and network technologies.

Adrien Lebre Dr. Adrien Lebre is a full time researcher at Inria (on leave from an Ass. Prof. position at the Ecole des Mines de Nantes) He received his Ph.D. from Grenoble Institute of Technologies in September 2006 and his M.S. degree from the University of Grenoble in 2002. His research interests are distributed and Internet computing. Since 2011, he is member of the architect board of Grid'5000. Dr. Adrien Lebre has taken part to several program committees of conferences and workshops.

Laurent Pouilloux Dr. Laurent Pouilloux is a research engineer from INRIA, also working inside the LIP laboratory at ENS Lyon, France. He received his Ph.D. at Institut de Physique du Globe de Paris in 2007 and has a strong experience in teaching computer science. He is mainly involved in helping Grid'5000 users to setup large-scale experiments, especially for virtual machines deployment and cloud computing experiments automation.