OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author's version published in: http://oatao.univ-toulouse.fr/22295

Official URL:
https://doi.org/10.1016/j.future.2015.11.027

To cite this version:
Tchana, Alain-Bouzaïde and De Palma, Noel and Safieddine, Ibrahim and Hagimont, Daniel Software consolidation as an efficient energy and cost saving solution. (2016) Future Generation Computer Systems, 58. 1-12. ISSN 0167-739X.

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr
Software consolidation as an efficient energy and cost saving solution

Alain Tchana\textsuperscript{a,\textdagger}, Noel De Palma\textsuperscript{b}, Ibrahim Safieddine\textsuperscript{b}, Daniel Hagimont\textsuperscript{a}

\textsuperscript{a} University of Toulouse, Toulouse, France
\textsuperscript{b} University of Grenoble Alpes, Grenoble, France

\textbf{Abstract}

Virtual machines (VM) are used in cloud computing environments to isolate different software. They also support live migration, and thus dynamic VM consolidation. This possibility can be used to reduce power consumption in the cloud. However, consolidation in cloud environments is limited due to reliance on VMs, mainly due to their memory overhead. For instance, over a 4-month period in a real cloud located in Grenoble (France), we observed that 805 VMs used less than 12\% of the CPU (of the active physical machines). This paper presents a solution introducing dynamic software consolidation. Software consolidation makes it possible to dynamically collocate several software applications on the same VM to reduce the number of VMs used. This approach can be combined with VM consolidation which collocates multiple VMs on a reduced number of physical machines. Software consolidation can be used in a private cloud to reduce power consumption, or by a client of a public cloud to reduce the number of VMs used, thus reducing costs. The solution was tested with a cloud hosting JMS messaging and Internet servers. The evaluations were performed using both the SPECjms2007 benchmark and an enterprise LAMP benchmark on both a VMware private cloud and Amazon EC2 public cloud. The results show that our approach can reduce the energy consumed in our private cloud by about 40\% and the charge for VMs on Amazon EC2 by about 40.5\%.

1. Introduction

Context and scope

In recent years, cloud computing has emerged as one of the best solutions to host applications for companies or individual users. For these cloud customers (hereafter called clients), its pay-per-use model reduces the cost compared to using internal IT resources. For cloud providers (hereafter called providers) one of the main challenges is limiting energy consumption in their data centers.

In 2010, for example, data centers consumed approximately 1.1\%–1.5\% of the world’s energy [1]. Energy consumption can be minimized by limiting the number of active physical machines (PM) through sharing the same PM between several software applications and providing dynamic software consolidation (filling unused resources by grouping software). This helps to balance the variable workload (Fig. 1 top presents an example of workload variation at Facebook) due to the departure of some software.

In this paper, we considered an SaaS-based cloud model, such as RightScale [4]. This type of cloud provides a fully customizable environment, allowing clients, e.g. companies, to focus on applications. The SaaS provider offers a software catalog. Clients can select an application and request its start in a virtualized data center. The data center may belong either to the SaaS provider (private cloud), or be part of a public cloud; alternatively it can be a mixture of the
two (hybrid cloud). The SaaS provider is responsible for managing the clients’ software (scalability, highly-available, failover, etc.) while efficiently managing resources to reduce data center costs: power consumption when relying on its own private cloud, or resource charged for when using a public cloud.

Problem and approach

Advances in virtualization make transparent dynamic consolidation possible in the cloud. Based on this technology, the cloud runs each software application on a separate virtual machine (VM). Many studies [5–8] have described algorithms providing software consolidation through the consolidation of VMs. However, this approach is not sufficient since an infinite number of VMs cannot be packed into a single PM, even when the VMs are underused and the PM has sufficient computation power. Indeed, as argued by [9], VM packing is limited by memory. In this paper, we therefore propose a solution consolidating software onto VMs. This solution is complementary to VM consolidation. Rather than dedicating one VM to each software, we propose that the same VM be shared between several software applications. This will fill the gaps remaining inside the VM, as mentioned earlier. Fig. 2 illustrates the benefits of this solution for VMs which are already at the minimum size allowed by the cloud. Using our strategy frees 2 PMs, while VMs consolidation alone only frees 1 PM. This strategy also reduces the total number of VMs (from 4 to 2 in the illustration). This is very important in a commercial cloud to reduce the charge for VMs. Fig. 1 bottom shows the average CPU usage by 805 VMs running on 66 PMs in a real virtualized cloud located in Grenoble (France) over 4 months. Each peak on the curve represents a significant variation in workload. For each VM, less than 12% of the CPU is used, but by applying our approach a single VM can host the workload of 8 VMs. This reduces the number of VMs running from 805 to about 101, and the number of active PMs from 66 to 9.

Contributions

Software consolidation raises two main challenges that need to be addressed:

- Software isolation. Isolation ensures that if a software application fails it does not compromise the execution of another software application, it also stops software from “stealing” the resources allocated to another application.
- Software migration. Migration involves moving software from its current node to another node without interrupting the service offered by the software, and while avoiding Service Level Agreement (SLA) violations on the migrated software.

In this paper we focus only on the live migration and consolidation mechanisms. For software isolation, we rely on Docker [10]. Docker can package an application in a virtual container, that runs processes in isolation. We present a solution to consolidate software on VMs (Section 2) based on a Constraints Programming (CP) solver. The gain of our approach is modeled in terms of power and cost savings, while limiting the consolidation-related risk to performance. The genericity of the solution allows the integration of a range of live software migration mechanisms since this operation is specific to software. We present a sample migration for JMS messaging servers and LAMP servers (Section 3) which are commonly and widely deployed in the cloud. We evaluated our approach using the SPECjms2007 benchmark [11] and an enterprise Internet application benchmark (Section 4) in the context of an SaaS offering messaging and Internet software on a private VMware cloud in our laboratory and on the Amazon EC2 cloud. These evaluations showed that: (1) our approach results in reduced power consumption and costs; and (2) the efficient live migration algorithms implemented for JMS messaging and Internet web servers are viable. For the specific workload assessed, our solution reduces the electricity consumption in our private cloud by about 40% when software consolidation is combined with VM consolidation. Running the same workload on Amazon EC2 leads to a reduction in VMs charged of about 40.5%. The paper ends by presenting related work in Section 5; a discussion about the usability of the solution is provided in Section 6; and a conclusion is provided in Section 7.

2. Software consolidation

Like VMs, software consolidation is an NP-hard [12] problem. This section presents a solution that allows software consolidation in the context of an SaaS platform.
2.1. Solution overview

We focus this overview on software consolidation and migration. VM placement at start time is part of the consolidation problem. Fig. 3 presents the key components of the model system studied. QuotaComputer determines the amount of resources required by each software application, it uses Docker containers [10] for isolation. Each container has a different IP address, thus two applications using the same port can run in the same VM. MonitoringEngine is responsible for gathering statistics for both VM and software from all MonitoringAgents. The ConsolidationManager implements an online, reactive software consolidation algorithm which acts as an infinite loop. It periodically:

1. gets VMs and software status (quota consumption, which is an average of the most recent values) from the MonitoringEngine.

2. checks if there are software applications which need more resources and provides for them (relocation algorithm described in Algorithm 1).
3. computes software assignment on VMs to minimize the number of VMs required to support all the software running. It also computes the reconfiguration plan (a set of software migrations) that must be performed for the ideal assignment to be achieved.
4. computes software assignment on VMs to minimize the number of VMs required to support all the software running. It also computes the reconfiguration plan (a set of software migrations) that must be performed for the ideal assignment to be achieved.
5. and finally, runs (through the LocalManager) the reconfiguration plan.

At the end of each loop, VMs not running any software are terminated, either immediately in the case of a private cloud, or when its uptime is close to a multiple of $\Theta$ (the payment time unit) in a public cloud. In the latter case, a timer is started for each VM to be terminated so that it stops before a new payment time unit starts. The timer is disabled when VM is eligible to host a running application (which can be newly-installed or relocated).

Before presenting our solution in detail, there follows a list of the notations used:

- $S = \{S_1, S_2, \ldots, S_m\}$ is the set of software types offered by the cloud.
- $S_i = \{S_1, \ldots, S_i\}$ is the set of software, instances of which are co-locate-able with an instance of $S_i$. This can be used to protect sensitive software from other potentially dangerous applications.
- For each VM $vm_i$, we consider three types of resources: cpu ($vm_i^c$), memory ($vm_i^m$) and IO bandwidth ($vm_i^b$).
- $vm_i^c$ is the cost of running the VM for a payment time unit $\Theta$. This is considered when the SaaS is placed on a public cloud.
- $vm_i^b$ is the start time of $vm_i$.
- An instantiated VM is assigned an identifier (an integer), $s_i^m$ is the identifier of the VM running software $s_i$.
- len($vm_i$) is the number of software applications on $vm_i$.
- Like VMware does for VMs, we consider two levels of resource reservation for a software: the minimum quota and the maximum quota. $s_i^b_{\text{min}}$ and $s_i^b_{\text{max}}$ are, respectively, the minimum
and the maximum cpu (or memory or IO) quota. The software starts with \( s_i^{\text{min}} \) and increases stepwise until it reaches \( s_i^{\text{max}} \). \( s_i^{\text{cur}} \) denotes the current quota. Note that * is u, m or o.

- \( s_i^{\text{cur}} \) is the acceptable service degradation threshold defined at start time for software \( s_i \). It corresponds to its SLA.

The relocation algorithm, Algorithm 1, checks if the current resources available for each software application are insufficient, excessive or sufficient. If it is insufficient, the software acquires more resources within its maximum quota. This operation can cause the software to be relocated to another VM (an existing one or a new one). On the other hand, if the software is wasting resources, its quota is reduced; the algorithm includes a clause to avoid the frequent transitions (yo-yo effect). The choice of the destination VM (on which software is to be relocated) does not need to be optimal. Indeed, the consolidation engine will correct the placement. This will be discussed in the next section, where the formalization of the software placement problem as a Constraint Satisfaction Problem (CSP) is presented along with how we used and optimized the Choco CSP solver [13] to resolve software placement problems.

2.2. Software placement as a CSP

**Definition.** A CSP [14], \( C \), is a set of constraints, \( \Delta \), acting on a set of variables, \( \Delta = \{A_1, A_2, \ldots, A_n\} \), each of which has a finite domain of possible values, \( D_i \). A solution to \( L \) is an instantiation of all of the variables in \( L \) such that all of the constraints in \( C \) are satisfied.

We used the Choco CP library [13] to solve CSP. Choco aims to minimize or maximize the value of a single variable, while respecting a CSP definition. To do this, it uses an exhaustive search based on a depth-first search. We used two CSPs to resolve the software consolidation problem. The first CSP was used to determine the minimum number of VMs \( n_{\text{new}} \) needed to run all software; we call this the MinVMToUse problem. But \( n_{\text{new}} \) can be provided by several configurations (software mapping onto VMs). Therefore, the second CSP chooses the appropriate configuration and generates the reconfiguration plan to reach that configuration; this is called the RightConfiguration problem. We modeled these problems as a mixed-integer non-linear optimization problem. The inputs are a list of VMs with their total resources, a list of software (for each VM) with their current resource quota and status (service level they provide).

The co-location of applications is defined by the SaaS provider. The co-location constraints are obtained using calibration and benchmark tests. The calibration can be automated using self-benchmarking tools (e.g., CLIF [15]).

2.2.1. The MinVMToUse problem

If no software deployment request has been submitted, the number of VMs in use after application of the ConsolidationManager should decrease or remain the same. This should be done while avoiding resource over commitment. This is expressed in the following inequality:

\[
\sum s_j^{\text{max}} \leq \sum v_m^i \land \sum s_j^{\text{cur}} \leq \sum v_m^i, \quad \forall \ VM \ v_m^i, \tag{1}
\]

where \( s_i^{\text{max}} = i \) and \( v_m^i \) is the resource capacity of the VM.

We also allow the user to specify co-location requirements for each software. The following equation expresses that:

\[
| s_i^{\text{min}} - s_j^{\text{min}} | + \text{Col}(s_i, s_j) \neq 0, \quad \forall \text{pair of software } (s_i, s_j), \quad \tag{2}
\]

\[
\text{Col}(s_i, s_j) = \begin{cases} 
1 & \text{if } s_i \text{ and } s_j \text{ are collocate-able} \\
0 & \text{otherwise}
\end{cases}
\]

The variable \( X \) minimizing the number of VMs is defined as follows:

\[
X = \sum (v_m^i) \text{if } (\forall v_m^i > 0) \text{ or } (\exists v_m^i = 0) \tag{3}
\]

2.2.2. Speeding up the consolidation process

We made some improvements to the consolidation process to reduce the solver execution time. First, we reduced the search domain for \( X \) by bounding it. In the best case, the minimum number of VMs is the sum of the resource quotas needed by all the software divided by the resource capacity of the biggest VM (we choose the most restrictive resource type). In the worst case, there will be no consolidation. This improvement is formalized as follows:
max\left(\frac{\sum_{i} s_{cur}^{i}}{\max\left(vm_{m}^{i}\right)}, \frac{\sum_{i} s_{cur}^{i}}{\max\left(vm_{m}^{i}\right)}, \frac{\sum_{i} s_{cur}^{i}}{\max\left(vm_{m}^{i}\right)}\right) \leq X \leq n,  \\
(4)

where \(n\) is the current number of VMs.

Second, some VMs or software may be equivalent in terms of resources or co-location constraints. If the resources offered by a VM, \(vm_{i}\), are insufficient to host software \(s_{j}\), then they are also insufficient to host any software \(s_{k}\) which has the same requirements. In addition, software \(s_{j}\) cannot be hosted by any other VM \(vm_{m}\) having the same characteristics as \(vm_{i}\). With regard to the co-location constraint, if a VM, \(vm_{i}\), runs software \(s_{j}\) which cannot be collocated with software \(s_{k}\), then \(vm_{m}\) cannot host any software of the same type as \(s_{j}\).

2.2.3. The RightConfiguration problem

For correct configuration, the solver only considers configurations using the number of VMs determined by the first problem. The reconfiguration operation likely to affect the software SLA is live migration. The impact of this process could be a degradation of the service offered by the migrated software. Three factors affect live migration: network utilization, remaining computation power on both source and destination VM, and efficiency of the implementation of the live migration itself. Considering this, we call \(s_{i}^{\Delta}\) the function calculating the impact of migrating software \(s_{i}\) for a given triplet of factors. Thus, if \(s_{i}^{\Delta}\) represents the current service level provided by \(s_{i}\) before migration, then \(s_{i}^{\Delta} s_{i}^{\prime}\) is the service level during migration. We define the cost of migrating a software \(s_{i}\) as \(s_{i}^{\Delta} = s_{i}^{\prime} - s_{i}^{\Delta} \times s_{i}^{\prime}\). The correct configuration is the one minimizing \(K\),

\[
K = \sum_{i} s_{i}^{\Delta}; \quad \forall \text {software } s_{i} \text{ to be migrated}
\]

while avoiding SLA violations:

\[
s_{i}^{\prime} s_{i}^{\prime} < s_{i}^{\Delta}, \quad \forall \text {software } s_{i} \text{ to be migrated}.
\]

3. Use cases

This work was conducted conjointly with two industrial groups: Scale Agent and Eolas. The former provides an implementation of the JMS specification, while the latter is an SaaS provider offering Internet services. We used our solution to manage an SaaS offering both a messaging service (such as IronMQ [16] and AmazonSNS [17]) provided by Joram [18] software, and an Internet service based on a LAMP architecture. This section presents the two use cases and the migration algorithms implemented. Migrating a running software serving requests (which is the case for both JMS and Internet servers) raises two main challenges that we had to address:

- (C1) Avoid loss of requests and state during migration.
- (C2) Make the migrated software available and accessible on the destination node after migration. This should be transparent for the clients.

3.1. JMS messaging servers

3.1.1. Overview of the messaging software Joram

Joram incorporates a 100% Java implementation of JMS 1.1 (Java Message Service) specification. It provides access to a truly distributed MOM (Message Oriented Middleware). Messages are handled through specific data structures called destinations: queue and topic. Fig. 4 presents how an application based on Joram functions. To build a Joram application, the first step is to create destinations, references of which must then be registered to a directory (commonly the JNDI). Destinations are hosted by Joram servers, and accessible using their IP addresses or DNS names. For our use case, we assume that clients only use the DNS name of the Joram servers in interactions. All Joram servers are accessible via another Joram server. This ensures that all messages will be delivered within a given time even when the destination server is out for a short time. This creates an automatic recovery feature for Joram applications.

3.1.2. Live migration of a Joram server

Joram ensures that any message will reach its addressee within a configurable time window. We relied on this feature to complete the initial part of the first challenge (C1). For the second part of (C1), at runtime a Joram server keeps a persistence basis containing its entire state: processing messages, messages in transit, and processed messages. Therefore, a Joram server can be made available with the same state on the destination node by copying this basis from the source node to the destination node. With regard to (C2), in contrast to live migration of VMs, where the migrated VM keeps its IP address on the destination node, migrating software results in a new IP address (the IP address of the destination node). How can remote clients be transparently informed of this new address? In our system this is resolved by forcing clients to use the DNS name when dealing with the Joram servers. Thus, the accessibility of the migrated server is provided by (1) dynamically updating the DNS server and (2) reconfiguring the JMS client to the DNS server. This is transparent to the client because the JMS client is implemented to automatically resolve new addresses after several attempts. Algorithm 2 summarizes the live migration process described above, the “naive” migration algorithm. As we show in the evaluation (Section 4.2), immediate copy of the persistence basis can have an important impact on the service offered by the migrated Joram server when this file is very large. To avoid this problem, we have optimized the algorithm to transmit the log file block by block to the destination node (Algorithm 3). This optimization was inspired by the copy-on-write mechanism used for live VM migration. We customized the Joram implementation to dynamically integrate and evolve its state at runtime when receiving new persistence information. A timer, which is triggered at the beginning of the migration process, ends the copy to limit the duration of the whole process. This optimization is currently being integrated into the official implementation of Joram on the OW2 [18] open source platform.

3.2. LAMP servers

3.2.1. Overview

Many Internet services provided in the cloud are based on a LAMP architecture. This is a set of Linux machines running Apache/PHP servers linked to MySQL databases. Eolas SaaS is based on this architecture. In addition to this architecture, it uses a HAProxy loadbalancer in front of Apache servers when several Apache servers are needed. The MySQL-Proxy loadbalancer is also used in front of a set of MySQL database servers. To cope with sessions lost when an Apache server fails, Eolas stores all user sessions on an NFS server. Our second use case follows this architecture, as summarized in Fig. 5.

3.2.2. Live migration of LAMP servers

We present only the migration algorithm for the Apache server. [19] proposes a live migration algorithm for MySQL servers which could be easily integrated into our solution. It must be remembered that all Apache sessions are stored on the NFS server; this facilitates conservation of its state after migration. The migration process is summarized by Algorithm 4.
4. Evaluations

We evaluated our solution to show the benefits of software consolidation on top of VM consolidation. These benefits are shown in terms of energy and cost savings. The efficiency and scalability of CSP-based consolidation methods were evaluated in [6,19]. As mentioned in the previous section, the SaaS we considered offers two applications: a JMS messaging application (with Joram) and a web application (with LAMP). Before assessing the energy and cost savings, we first evaluated the migration algorithms implemented for the different software.
4.1. Testbed overview

4.1.1. The cloud infrastructure

The cloud testbed integrates both a private and a public platform. Our private cloud is a part of the Eolas data center. It is composed of 8 DELL PowerEdge R510 equipped with Xeon E5645 2.40 Ghz processors (one with a 12-core CPU, and the others with 8-core CPU), 32 Gb memory and 2 NICs at 1 Gbps. They are connected through a gigabyte network switch. The virtualized layer is provided by VMware VCenter 5.1.0 (ESXi 5) with the VM consolidation module DRS/DPM [20] enabled: a PM for the VCenter, a PM with an NFS server to host VM images and user sessions, and 5 PMs as ESXi to host VMs. The last PM hosts our system (including the DNS server) and the agents simulating the Joram and web server users. The cloud provides a single type of VM: 1 vcpu running at 2.4 GHz and 1 Gb memory. The public used was Amazon EC2 in the M1, medium VM, configuration.

4.1.2. Benchmarks

SPECjms2007 [11] was used to bench the Joram servers. SPECjms2007 is the industry-standard benchmark for evaluating the performance of enterprise MOM servers based on JMS. SPECjms2007 models the supply chain for a chain of supermarkets. The scenario offers a natural scaling of the workload, e.g. by adjusting the number of supermarkets (horizontal) or by adapting the number of products sold per supermarket (vertical). Its configuration parameter base sets the scalability factor. The scenario includes seven interactions. Each interaction can use several types of destinations (queues or topics) and run over several steps. A scenario is organized in 3 phases: warm-up period (scaling up), measurement period (constant message injection), and drain period (last messages treated before the end of the benchmark). SPECjms2007 defines the SLA for this scenario as follows: 90% of messages should be delivered within 5 s. For our experiments we used vertical scalability with base configured to 26 (generating up to 135 destinations). For each type of interaction, a single Joram server hosts all of its destinations. Thus, 7 servers are needed to run a scenario. Among these servers, we consider the one hosting destinations for the statistics interaction (the HQ_SMStatsQ queue) as the most representative (hereafter called the indicator) for a scenario. This server is, in fact, the most solicited. Accordingly, only the results relating to the HQ_SMStatsQ destination will be presented. To simplify reading, we express a scenario as follows: warm-up#measurement#drain#nOfVMUSeD. For all experiments, each Joram server was configured to use one-tenth of a VM as the minimum quota and a quarter as the maximum.

The second use case was based on real traces of the Internet service offered by the Eolas SaaS. We played the traces with the Neoload [21] load injector toolkit to submit the workload to our LAMP servers. However, a synthetic load profile was used to evaluate the impact of migration. Traces were only used for the overall evaluation.

4.2. Impact of migrating a Joram server

To evaluate the impact of migrating a Joram server, we ran a set of experiments (on both a private and a public cloud) with two scenarios 300#1200#0#2: one requiring migration and the other without migration. Migration consists of moving an indicator from $vm_1$ to $vm_2$. Although the non-migrated Joram servers on the source and destination VMs were not affected by the migration, the migrated server was degraded during the migration, as shown in Fig. 6. Thus, message delivery time on the HQ_SMStatsQ destination was adversely affected by migration on the private cloud using the “naive” algorithm, with messages being delivered within about 7 s during migration while the migration itself takes about 8 s.

The benefit of the optimization of the migration algorithm was evaluated by running the same experiment with optimization (WO) and without optimization (WO) while varying the migration time (which implies copying different Joram server log file sizes). The process was assessed in the middle of the ramp-up period (MRU), at the beginning of the measurement period (BM), 3 min after the measurement period (3AM), and in the middle of the measurement period (MM) (Fig. 7). Varying the migration time during the experiment reveals how the two algorithms cope with the increasing size of the migrated Joram server log file. The top panel shows the duration of the migration process while the lower panel shows the average delivery time for messages during the migration. We can see that the optimized algorithm results in an almost constant migration duration whereas the naive algorithm results in an exponential increase in duration of migration. This is caused by the fact that the migrated Joram server has to load its
reduce power consumption. The following linear power consumption model of a PM at time \( t \) is widely used in the literature [22]:

\[
P_i(U(t)) = p_i^{\text{min}} + (p_i^{\text{max}} - p_i^{\text{min}}) \times U_i(t)
\]

\[
P_i^{\text{max}} \text{ is the maximum power consumed by a PM when it is fully used, } p_i^{\text{min}} \text{ is the power consumption when it is idle, and } U_i \text{ is the CPU utilization level. } E \text{ is the total energy consumed by the cloud infrastructure. } P_i(U(t)) \text{ increases with } U_i(t) \text{, which depends on the number of VMs and software applications running on the PM:}
\]

\[
U(t) = \sum_{i=1}^{\text{len}(\text{PM}_i)} p_i(U_i(t))
\]

\[
\text{with } P(U_i(t)) = \sum_{j=1}^{\text{len}(v_m_i)} s_i^{j,n} + \Omega_i \text{ such that } s_i^{j,m} = i
\]

\( \text{len}(\text{PM}_i) \) is the number of VMs on \( \text{PM}_i \), and \( \Omega \) is the VM overhead. According to Eq. (9), relocating a service from one VM to another VM is beneficial in terms of reducing power consumption when it leads to a VM termination. In the worst case, this gain is

\[
\int_0^t (p_i^{\text{max}} - p_i^{\text{min}}) \times \Omega dx
\]

in the best case (the VM running alone on its PM), it is the power consumed by a PM which can be stopped.

The benefit of reducing the number of VMs from \( n \) to \( n' \) is given by the following formula:

\[
\sum_{i=1}^{\text{nbPM} - \text{nbPM}'} E_i + (n - n') \int_0^t (p_i^{\text{max}} - p_i^{\text{min}}) \times \Omega dx
\]

where \( E_i \) is the power which would have been consumed by the PM \( i \) if not switched off; \( \text{nbPM} \) and \( \text{nbPM}' \) are, respectively, the number of PMs used to host \( n \) and \( n' \) VMs.

4.3. Impact of migrating an Apache server

To evaluate the impact of migrating an Apache server, we ran a set of experiments with different workloads. The migration consists of moving the Apache server from its initial VM to another VM (in our case the VM running the MySQL server). It must be remembered that taking the new location of the migrated Apache server into account requires reconfiguration and the reloading of the HAProxy load balancer. This is the most crucial step because no requests can be treated by the application during reloading. This could lead to lost requests. We performed a set of experiments while varying the number of Internet users to see if the number of requests lost depended on the workload (Fig. 8). These results show that with up to 25 simultaneous users (time \( t_1, t_2, \) and \( t_3 \)) no requests are lost during migration. The consolidation algorithm can be configured so that the system will discard migrations where requests have been lost. This strategy has been applied for the remainder of the experiments described here. Fig. 8 also shows that migration does not affect the application’s throughput or response time. Throughout the experiment, the migration time is almost constant, at about 40 s.

4.4. Power saving in the private cloud

4.4.1. Formalization

Software consolidation packs software onto a minimum number of VMs and terminates free VMs. This gives a greater VM consolidation capacity in the virtualization layer. Thus, if combined with a VM consolidation mechanism such as [6] or VMware DRS/PDM [20], the cloud infrastructure could turn off some PMs to reduce power consumption. The following linear power consumption model of a PM at time \( t \) is widely used in the literature [22]:

\[
P_i(U(t)) = p_i^{\text{min}} + (p_i^{\text{max}} - p_i^{\text{min}}) \times U_i(t)
\]

\[
E = \sum E_i \text{, where } E_i = \int_0^t P_i(U_i(x)) dx
\]

\( E \) is the total energy consumed by the cloud infrastructure. \( P_i(U(t)) \) increases with \( U_i(t) \), which depends on the number of VMs and software applications running on the PM:

\[
U(t) = \sum_{i=1}^{\text{len}(\text{PM}_i)} p_i(U_i(t))
\]

\[
\text{with } P_i(U_i(t)) = \sum_{j=1}^{\text{len}(v_m_i)} s_i^{j,n} + \Omega_i \text{ such that } s_i^{j,m} = i
\]

\( \text{len}(\text{PM}_i) \) is the number of VMs on \( \text{PM}_i \), and \( \Omega \) is the VM overhead. According to Eq. (9), relocating a service from one VM to another VM is beneficial in terms of reducing power consumption when it leads to a VM termination. In the worst case, this gain is

\[
\int_0^t (p_i^{\text{max}} - p_i^{\text{min}}) \times \Omega dx
\]

in the best case (the VM running alone on its PM), it is the power consumed by a PM which can be stopped.

The benefit of reducing the number of VMs from \( n \) to \( n' \) is given by the following formula:

\[
\sum_{i=1}^{\text{nbPM} - \text{nbPM}'} E_i + (n - n') \int_0^t (p_i^{\text{max}} - p_i^{\text{min}}) \times \Omega dx
\]

where \( E_i \) is the power which would have been consumed by the PM \( i \) if not switched off; \( \text{nbPM} \) and \( \text{nbPM}' \) are, respectively, the number of PMs used to host \( n \) and \( n' \) VMs.

4.4.2. Evaluation results

We simultaneously ran 15 SPECjms2007 and 6 LAMP scenarios (up to 37 VMs) in two situations. In the first situation (noted WSC (With Software Consolidation)) we ran the experiment with both software and VM consolidation enabled, while in the second situation (noted WOSC (Without Software Consolidation)) we disabled software consolidation (but maintained VM consolidation). The scenarios were configured to provide a varied workload over 30 h: a mix of constant, ascending and descending phases. Fig. 9 presents (1) the occupancy (in terms of the number of VMs) of each PM in the private cloud, and (2) the number of PMs in use during the 25 h of observation. We see that the first situation results in 3 PMs (PM2, PM3 and PM5) being freed, while 1 PM (PM2) was freed in the second situation. As formalized in the previous section, software consolidation definitively accelerates VM consolidation. The bottom right curve in Fig. 9 shows that this improvement represents an approximately 40% power saving with this particular workload.
4.5. Cost saving in a public cloud

4.5.1. Formalization

When relying on a commercial cloud, the SaaS provider is charged as a function of the total number of VMs used. Therefore, minimizing the invoice is equivalent to the MinVMToUse problem. The cost of using the public cloud at time $t\theta$ is given by $\text{TotalCost}(t \theta)$. Let $\text{Old}(t-1)\theta$ be the list of VMs existing since time $(t-1)\theta$, $\text{New}(t-1)\theta$ be the list of new VMs started after time $(t-1)\theta$ but before $t\theta$, and

$$\text{Cost}(\text{listOfVMs}) = \sum_{i=1}^{\text{listOfVMs}} \text{vm}_i^\theta$$

be the cost of running the list of VMs, listOfVMs, for the payment time unit $\theta$. Then

$$\text{Cost}(t \theta) = \text{Cost}(\text{Old}(t-1)\theta \cup \text{New}(t-1)\theta)$$

with $\text{Old}(0) = \emptyset$.

$\text{New}(0) = \{\text{the set of VMs started for the first time by the cloud}\}$

Therefore

$$\text{TotalCost}(t \theta) = \sum_{i=1}^{t \theta} \text{Cost}(i \theta)$$

$\text{Cost}(t \theta)$ is the additional cost of using the cloud from time $(t-1)\theta$ to time $t\theta$. This cost is kept to a minimum since $\text{Old}(t-1)\theta \cup \text{New}(t-1)\theta$ is minimized by the MinVMToUse problem. Therefore, $\text{TotalCost}(t \theta)$ is the minimum cost of using the cloud at time $t\theta$.

4.5.2. Evaluation results

We repeated the previous experiments on the private cloud with VMs configured to run for an hour (before termination because they were empty) on Amazon EC2. We used M1, medium VMs instances, which are charged at $0.12$/h per VM per hour. Fig. 10 presents the total number of VMs used over the 25 h of observation, and the total cost of the experiments. The number of VMs is seen to drastically decrease thanks to software consolidation, resulting in an approximate 40.5% saving: from about $1300 (without software consolidation) to $800 (with software consolidation).

5. Related work

Memory footprint improvements. Significant research has been devoted to improving workload consolidation in data centers. Some studies have investigated reducing the VM memory footprint to increase the VMs' consolidation, when a VM is dedicated to a single software. Among these, memory compression and memory over commitment [23–26] are very promising. In the same vein, [27] extends the VM ballooning technique to software to increase the density of software collocation on the same VM; [28] presented a similar approach. [28] presented VSwap, a guest-agnostic solution to reduce the effect of memory ballooning. Another study, [30], presents an energy-aware workload consolidation framework at the kernel level. The power and performance prediction models presented in this work were used to consolidate a set of processes as a single Graphic Processing Unit (GPU) workload. Xen offers
what is called "stub domain." This is a lightweight VM which requires very few memory (about 32 MB) for its execution. As our solution, all these works try to minimize the footprint of a VM in order to increase the number of VMs that can be collocated on top of the same physical machine. Therefore, they result to the same result in terms of energy saving. However, they do not minimize the total number of VMs as we do in order to reduce VMs charged for the clients when considering of a commercial cloud.

**Impact of VM consolidation.** Several research [31,32] have investigated the impact of consolidating several VMs on top of the same host. This situation leads to resource contention/interference which is the main origin of performance degradation. [33] studies the problem by characterizing workloads which can be collocated without enough interference. [34] studied interference when collocating MapReduce applications. It proposes a tasks scheduling strategy based on a performance prediction model in order to minimize the impact of co-locating competitive tasks. [35] studied the effects of collocating different types of VMs under various VM to core placement schemes in order to discover the optimal placement for performance. We perform in this paper a number of experiments in the same direction. [36] proposed a software probe for detecting contention because hardware platform specific counters (usually used) are not always sufficient. [37] characterized IO performance to study the impact of collocating several software applications running on separate VMs on the same PM. The conclusion of this paper is useful for users of our framework since it could be used to define collocated software. [35] also studied the impact of collocating VMs on the same PM and proposed an interference metric and regression model. Along the same lines, [38] motivated the need to study workload consolidation and provided a VM consolidation framework. The objective of their system was to organize all the VMs (or sets of users) in the cloud in terms of groups of VMs that can share the same PM with negligible interference. [39] analyzed the trace of a real data center and proposed a workload consolidation algorithm which avoids collocation of non-collocate-able software on the same PM. Its consolidation algorithm also avoids SLA violations when a peak load occurs during migration. This was feasible in [39] because it studied a trace. [40] presents a technique to predict performance interference due to sharing a processor cache. This technique works on current processor architectures and predicts degradation for any possible placement. It can therefore be used to select the most efficient consolidation pattern. The solution presented here takes the impact of consolidation into account by organizing software which would severely compete for the same resources. The solution we present in this paper takes into consideration the impact of consolidating competitive workload onto the same VM. The responsibility is given to the provider. Recall that we consider in this paper SaaS hosting centers. In this context, the provider well knows the applications he provides. Thus, he knows (e.g. through experiments) which application or part of an application can be collocated with which one, knowing that our solution provides a way to consider these constraints.

**VM consolidation algorithms.** The research community made an important contribution to workload consolidation through VM consolidation using various heuristic algorithms. In our previous work [5], we proposed a couple of this sort of VM relocation and collocation algorithms. Likewise, [41] proposed a simple VM consolidation algorithm which is similar to the First-Fit Decreasing (FFD) heuristic [42], while [43] customized the FFD algorithm to integrate the cost of live VM migration. [44] studied a set of heuristics algorithms and proposed new versions of heuristics when the VM migration cost was taken into account. In these new versions, several experiments on real workloads were performed. [45] presented a VM consolidation framework considering CPU, memory, and IO resources. Its colocation algorithm also considered network communication intensity between VMs, and PM temperature. The VM placement algorithm, a heuristic bin-packing algorithm, is not described. The network communication intensity described by Beloglazov et al. [45] could be integrated into our framework, but the temperature monitor is not feasible since we do not measure the PM temperature. [46] presents an adaptive heuristic for VM consolidation based on analysis of past VM resource usage. A set of formalizations for the VM placement problem is also provided. [47] also treats the VMs consolidation problem using a heuristic algorithm which minimizes the number of live migrations in the reconfiguration process.

---

2 http://wiki.xen.org/wiki/StubDom
plan. An SLA-aware VMs consolidation system is presented in [41]. Like with our proposal, it formalizes the problem of minimizing the operating cost for a private cloud while also minimizing SLA violations for services offered by software. Our formalization can be extended by considering this work. [48] presents an energy-aware VM placement and a consolidation algorithm. These are specific to the snooze cloud platform [48]. The consolidation algorithm is restricted to use in a homogeneous environment and reuses elements from [47]. [49] consolidates tasks onto VMs using a First-Fit algorithm. Only the basic formalization of the problem is described, and no details are given about the system. [50] presents a VM consolidation strategy based on a predictive approach. Since the placement problem is NP-hard, it is not possible to develop a solution running within an acceptable time. [51] presents DejaVu, a consolidation system which takes into consideration the interference between consolidated VMs. Based on hardware counters, it proposes a metric for characterizing workloads which are collocatable. In this paper, we do not focus on VM consolidation. We bring the same idea at software level (software within VMs). Therefore, any VM consolidation algorithm presented in this section can be applied to software consolidation. In this paper, we base on a solver to resolve the problem.

Software consolidation. The main problem with previous solutions is that they are limited by the footprint of the VMs consolidated (they are all operating systems). Execution of a VM requires a set of minimum resources, even if the application it runs is idle. Thus, we propose a solution which dynamically packs software into VMs to effectively use the overall VM resources while respecting the individual requirements of the different software applications. With current knowledge, [19] is the only previous work that studies dynamic software consolidation on the same OS; however, it does not rely on VMs. [19] focuses on the MySQL database software and provides a live migration algorithm for that. This algorithm can be plugged into our framework. [19] (as well as Entropy [6]) describes a consolidation algorithm based on a Constraint Satisfaction Problem (CSP) [14]. Thus, no previous study has investigated software consolidation onto VMs. In this paper, we developed a working prototype and showed that it can achieve high VM utilization to provide cost- and power-saving benefits.

6. Discussion

The work presented in this paper does not re-invent the wheel. We propose an effective transposition of VM consolidation (a widely and commonly approved solution for energy saving in a IaaS) into software consolidation. In addition to energy saving, we show how this solution can be benefited (in terms of cost saving) for public cloud customers (clients). As we argued in the Introduction, a software collocation solution must provide isolation mechanisms, which are the main advantages of VMs. We claim in this paper that in some situations, the need of a full/strong isolation as provided by VMs is not necessary. Some lightweight solutions such as cgroup or chroot are sometimes sufficient. For instance, when we consider a SaaS hosting center where the provider well knows applications and is the only manager of the infrastructure, the probability to have troubleshooting coming from outside or from VMs is minimized. In this context, consolidating software on top of the same VM makes sense. Combining this with a traditional VM consolidation system will increase the number of powered-off machines (which leads to energy saving). Another use case concerns enterprises or individual users who want to deploy their application within a public commercial cloud. In this context, they need to minimize their VMs charged. This is achieved by reducing the number of active VMs. As for a SaaS provider, the enterprise or the individual user can rely on our solution to do that.

Live migration is another important mechanism that should be offered in order to support dynamic software consolidation. This could be considered (in point of view of the user of our framework) as the main drawback of our solution since it requires the user to provide for each application component the implementation of its live migration. This paper presents various live migration implementations for JMS (integrated within Joram [18] development branch and being tested for the coming release) and Internet services, which are among the most deployed applications on the internet. Also, the framework we introduce in this paper is built such a way that the integration of new live migration implementations (according to the considered applications) is facilitated. Evaluations results we have obtained both in our private cloud and Amazon EC2 show the viability of our solution.

7. Conclusion

In this paper we proposed a solution to consolidate software onto VMs to reduce power consumption in a private cloud and the number of VMs charged for in a public cloud. We focused on the algorithms for live migration and consolidation. Although the proposed solution can integrate other live software migration algorithms, we demonstrated that the algorithms were efficient for JMS messaging and web servers. The consolidation algorithm was inspired by Entropy, which treats VM consolidation based on a Constraint Satisfaction Problem (CSP) approach. Evaluations with realistic benchmarks on a messaging and web applications SaaS cloud showed that our solution (1) reduces the power consumed by our industrial cloud partner by about 40% when combined with VM consolidation, and (2) reduces the charge for VMs used on Amazon EC2 by about 40.5%. Future work will include extended analysis of how best to coordinate software consolidation on VMs with VM consolidation on physical machines in order to further improve power gains.

Acknowledgments

This work is supported by the French Fonds National pour la Societe Numerique (FSN) and Poles Minalogic, Systematic and SCS, through the FSN Open Cloudware project.

References
