Optimal resource allocation in a virtualized software aging platform with software rejuvenation

Javier Alonso, Íñigo Goiri, Jordi Guitart, Ricard Gavaldà and Jordi Torres

Universitat Politècnica de Catalunya and Barcelona Supercomputing Center, Jordi Girona 31, 08034 Barcelona, Spain

Email: {alonso, igoiri, jguitart, torres}@ac.upc.edu, gavalda@lsi.upc.edu

Abstract—Nowadays, virtualized platforms have become the most popular option to deploy complex enough services. The reason is that virtualization allows resource providers to increase resource utilization. Deployed services are expected to be always available, but these long-running services are especially sensitive to suffer from software aging phenomenon. This term refers to an accumulation of errors, which usually causes resource exhaustion, and eventually makes the service hang/crash. To counteract this phenomenon, a preventive approach to fault management, called software rejuvenation has been proposed.

In this paper, we propose a framework which provides transparent and predictive software rejuvenation to web services that suffer software aging on virtualized platforms, achieving high levels of availability. To exploit the provider resources, the framework also seeks to maximize the number of services running simultaneously on the platform, while guaranteeing the resources needed by each service.

I. INTRODUCTION

The growing popularity of virtualization technology offers the possibility that different services coexist in the same physical node, sharing the same physical resources, without disturbing each other [1], [2]. Virtualization technologies like XEN [3] or VMWare [4] allow us to deploy several virtual machines with different operating system and services on the top of the physical node, without taking into account the underlying hardware and the physical details. These capabilities have allowed the increase of resource utilization. Prior to the rise of virtualization, resource utilization ratio was only around 5% to 20%. This means that these new virtualized platforms reduce the IT infrastructure budget cost, making them the most popular option to deploy complex enough services.

Services deployed on virtualized platforms are expected to be always available. However, long-running applications such us web business services (i.e. Amazon store, eBay, etc..) are specially sensitive to suffer from software failures. In fact, currently, it is well known that unplanned service failures are mainly caused by software faults [5], [6]. One of the reasons for unplanned downtime caused by software faults is the software aging phenomenon. This term refers to an accumulation of errors, which usually causes resource exhaustion, and eventually makes the service hang/crash. This phenomenon is especially evident in long-running applications, where this accumulation of software bugs during a long period of time could cause unplanned outages.

Furthermore, these unplanned outages have an important impact on the company's revenues and reputation/trust: due to the economic impact on revenues that carries an outage and the discredit that accompanies unplanned outages, causing possible loss of future customers [7]. To counteract the software aging phenomenon, a preventive approach to fault management, called software rejuvenation has been proposed [8]. Software rejuvenation strategies can be divided into two basic categories: Time-based and Proactive/Predictive-based strategies. In time-based strategies, rejuvenation is applied regularly and at predetermined time intervals. In Proactive/Predictive rejuvenation, system metrics are continuously monitored and the rejuvenation action is triggered when a crash or system hang up due to software aging seem to approach.

This paper presents a framework which provides a transparent and predictive rejuvenation solution to web services on virtualized platform. It manages automatically (or with minimal human intervention) the virtualized platform to guarantee the services availability against software aging. The software rejuvenation solution is implemented and integrated transparently to the deployed service. Furthermore, to exploit the provider resources, the framework also seeks to maximize the number of services running simultaneously on the platform. At the same time, the framework ensures that every service deployed has the resources requested by the service owner. A mathematical model is also presented to achieve both goals. The novelty of our approach is the integration of software rejuvenation in the maximization of number of services problem. The mathematical model integrates both goals because the rejuvenation strategy consumes extra resources for short periods of time. This affects the effective resources available, impacting on the new service acceptance decisions. We present two different policies based on the same model to manage the virtualized platform.

The rest of the paper is organized as follows: Section II describes our framework and outlines the benefit of our approach; Section III discusses the mathematical reasoning behind the framework decisions. Section IV presents the experimental study setup used. Section V presents and discusses our experimental study results. Section VI presents the related work; and, finally, Section VII concludes the paper.

II. FRAMEWORK COMPONENTS DESCRIPTION

In this section, we present the main components of our framework: the *Service Container* and the *High Availability and the Resource Optimization Scheduler (HARO Scheduler)*. The *service container* is a logical element which wraps the service deployed and the transparent and predictive software rejuvenation components. On the other side, The *HARO Scheduler* is responsible of guaranteeing the availability of potential aging services and accept as many services as possible according to the platform resources and the virtualization middleware (i.e. Virtual Machine Monitor).

Figure 1 presents an overview of the components interaction.

A. Service container

The service container is a logical structure. Every service deployed on the platform has three elements (in a logical view) associated: High Availability Load Balancer (HA-LB), Aging Predictor (AgingPdr) and, accordingly with the service state a service replica. The service is deployed in its own virtual machine (ServiceVM). ServiceVM implements a monitoring agent, collecting the VM resource status like CPU, Memory, number of Threads or number of connections. These metrics are used to detect the software aging phenomenon and estimate the service time to crash. This estimation is conducted by
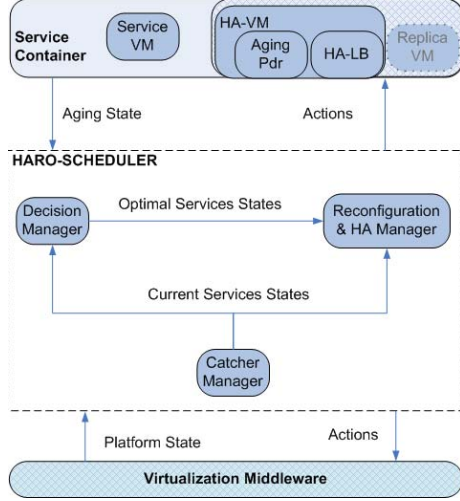
IEEE computer society

Fig. 1: Communication process between components

AgingPdr. It is integrated by a Machine Learning model trained to estimate the time to crash due to complex software aging phenomenon. The model used in our current approach is based on M5P Machine learning algorithm [9], which presents promising results in terms of accuracy prediction in software aging scenarios [10]. We note that the AgingPdr could be changed with new detection/estimation failures methods to maintain the effectiveness of the framework in new environments. AgingPdr is running jointly with HA-LB, in their own virtual machine (HA-VM). HA-LB is a 4-layer load balancer which manages the software rejuvenation process. The software rejuvenation solution is partially based in our previous studies [11], [12], showing the effectiveness of the approach. During the rejuvenation approach, a service replica (ReplicaVM) is needed. So, for a short period of time two copies of the same service are working simultaneously. This process, which is presented in detail below, guarantees that during the rejuvenation, any on-going or new request to the service is not missing.

B. High Availability and Resource Optimization Scheduler

The *HARO Scheduler* has the responsibility to take decisions about services based on the state of the platform. It also decides the correct moment to rejuvenate a service. Moreover, it allocates the services to maximize the number of services on the platform according to the resources available. The *HARO Scheduler* is composed of the following items: the *Catcher Manager*, the *Decision Manager* and the *Reconfiguration & HA Manager*.

Catcher Manager (CM). It is aware of all service deployed on the platform. The *CM* also receives the new service requests to deploy it on the platform. However, it does not decide if the request is accepted or not. So, *CM* maintains a list of running and waiting services (new requests).

The *CM* also maintains a list of the resources required by every service to run properly. These resources define the *service size*. They are basically: number of (virtual) CPUs, RAM memory, number of network interfaces, etc.... These values are important because the framework uses them to calculate how many services can run simultaneously in one physical node. This has sense since the virtualization middleware defines how many virtual resources can be assigned to one physical resource.

In order to calculate the size of a new service, we sum the resources requested by the service provider and the resources required by the software aging components (HA-VM). The size of a new service is the service container size, without ReplicaVM.

The current state of the platform collected and maintained by CM is sent to *Decision Manager* and *HARO Scheduler*.

Decision Manager (DM). It receives the data collected by *CM* and its main task is to calculate the maximum number of services that could be accepted by the infrastructure. The *DM* decides if a service could be accepted or not. The decision is taken based on the size of the service container (the sum of the *Service VM* and the *HA-VM*) and the current available resources on the platform. If it is needed to migrate running VMs to free space for new services, the *DM* indicates it. *DM* also prevents that every deployed service has the resources requested by the service owner. Moreover, it takes care to avoid violating the maximum number of VMs that one physical node could execute simultaenously. This limit is fixed by the physical resources available. This approach guarantees a minimum performance level, based on the resources requested by service owners.

On the other hand, the *DM* also decides if a service is suffering software aging and if a software rejuvenation has to be triggered. This decision is based on the time to crash (TTC) calculated by *AgingPrd* associated to every service and the threshold (time limit (TL)) defined by the system administrators. TL indicated how much time (i.e. in seconds) before the possible crash, the rejuvenation has to be triggered. The Time limit could be fixed for the whole platform or per each service individually. The software rejuvenation process requires creating a *ReplicaVM*. The *DM* also decides where (the physical node) the ReplicaVM must be created.

The data generated by CM and DM is represented using the following structure:

$$\widetilde{X} = \begin{pmatrix} \widetilde{X}_{0,0} & \widetilde{X}_{0,1} & \widetilde{X}_{0,2} \\ \widetilde{X}_{1,0} & \widetilde{X}_{1,1} & \widetilde{X}_{1,2} \\ \widetilde{X}_{2,0} & \widetilde{X}_{2,1} & \widetilde{X}_{2,2} \end{pmatrix} \widetilde{X}' = \begin{pmatrix} \widetilde{X}'_{0,0} & \widetilde{X}'_{0,1} & \widetilde{X}'_{0,2} \\ \widetilde{X}'_{1,0} & \widetilde{X}'_{1,1} & \widetilde{X}'_{1,2} \\ \widetilde{X}'_{2,0} & \widetilde{X}'_{2,1} & \widetilde{X}'_{2,2} \end{pmatrix}$$

$$W = \begin{pmatrix} W_0 & W_1 & W_2 & W'_0 & W'_1 & W'_2 \end{pmatrix}$$

where:

$$\begin{cases} \widetilde{X}_{i,j} = 1 & \text{if the VM j is in node i} \\ \widetilde{X}_{i,j} = 0 & \text{otherwise} \\ \widetilde{X}'_{i,j} = 1 & \text{if a replica of VM j is in node i} \\ \widetilde{X}'_{i,j} = 0 & \text{otherwise} \end{cases}$$

$W_j (= W'_j)$ represents the size of VM j (or replica)

The matrices $\widetilde{X}$ and $\widetilde{X}'$ represent the position of running/waiting services and *Replica VMs* respectively. The rows represent the physical nodes and columns the virtual machines. A running service is composed by two VMs (service VM and HA VM), so it is represented by two columns. Although, a service waiting to be deployed is represented by one column, composed of all-zeros. On the other side, a 1 in a column represents the physical node where the VM is running. W contains the *size* of all VMs. We note that the size of waiting services is the sum of the real *Service VM* and its *HA VM*, so the size of the full service container. The matrices notation is: $\widetilde{X}$ and $\widetilde{X}'$ for CM matrices. In order to differentiate them from DM matrices, DM matrices notation is X and X', following the same pattern like CM matrices presented earlier. This notation is used in the rest of the document.

Reconfiguration & HA Manager (RHAM). The data generated by *CM* and *DM* is sent to the *RHAM*. The *RHAM* basically conducts

a comparison between the current state of the platform ($\widetilde{X}$ and $\widetilde{X}'$ generated by the *CM*) and the optimal platform state (X and X' generated by the *DM*). The process is simple and the framework could take the decisions quickly, adapting the current state to the optimal state in a short period of time. Figure 2 presents the three main processes *RHAM* conducts:

– *Service Creation:* when the *RHAM* decides the creation of a new service, it is because at least one service is waiting to be deployed and there is enough space to allocate it. When a new service is deployed, transparently to the service owner, a *HA VM* is also created; creating two VMs. Figure 2 presents an example of how *RHAM* decides the creation of a new service. The all-zeros fourth column (oval dotted line) at $\widetilde{X}$ indicates service 4 is waiting to be deployed. *RHAM* compares the fourth column from $\widetilde{X}$ with the same column of X. The fourth column of X is non-all-zeros ($X_{2,4} = 1$, dot line circle). This indicates that *DM* concluded that the best position to deploy service (*Service VM* and *HA VM*) 4 is node 2.

– *Service Migration:* it is conducted using the well-known technology offered by almost all virtualization manufacturers: live-migration [13]. This technology allows us to migrate a running VM without any or minimal outage, from one physical node to another. It is important because we will use this mechanism to reallocate VMs, optimizing the resource usage. The *RHAM* decides a *Service Migration*, comparing four matrices. An example is presented in Figure 2, marked with black line circles. VM 3 is currently at node 1 ($\widetilde{X}_{1,3} = 1$), however, the *DM* marks that the optimal position for this VM is at node 3 ($X_{3,3} = 1$). The *RHAM* has to migrate VM 3 from physical node 1 to 3. The same process can be conducted with *Replica VMs* or even *HA VMs*.

– *Service Recovery:* it is triggered due to the TTC of one or a set of VMs have violated the threshold (Time Limit) defined by the system administrators per each service or per the whole platform. However, the *DM* has to decide what is the best place to create the replica. This process consumes extra resources to allocate the same number of services. *DM* (its mathematical model) integrates these extra VMs in the maximization function to reduce the impact of replicaVM creation. The replicaVM position is presented by matrix X'. Figure 2 presents an example (grey line circles). *DM* detects aging in VM 0 and it does not have a replica yet (all-zeros column of matrix $\widetilde{X}'$). *DM* decides creating a replica at node 2 ($X'_{2,0} = 1$).

C. Transparent predictive Software Rejuvenation Approach

The proactive software rejuvenation process is managed by *HA-LB* and it is divided into three phases presented in Figure 3.

1) Aging failure detection: the *HARO Scheduler* detects the Time To Crash (TTC) of the service (estimated by its *AgingPdr*) is higher than the threshold (Time Limit) defined by the administrators for this service. The prediction is conducted by M5P Machine Learning Algorithm [9]. It estimates the time to crash of the service based on the monitored service metrics. In [10], it is shown the effectiveness of this algorithm to estimate the time to crash of a web service under complex and undeterministic aging scenarios.

2) Workload migration process: the *RHAM* creates a *ReplicaVM* in the better physical node to maximize the space available in the platform. At the same time, it warns *HA-LB* that a recovery action over its monitored service started. *HA-LB* automatically migrates new requests to the replica service. The on-going requests being processed by faulty service are allowed to finish.

3) Faulty Service VM elimination: when *HA-LB* detects that its monitored service is empty (no requests under processing), *HA-LB*

sends the order to *RHAM* to remove the faulty *ServiceVM*. Then, the *ReplicaVM* becomes the new *ServiceVM*.
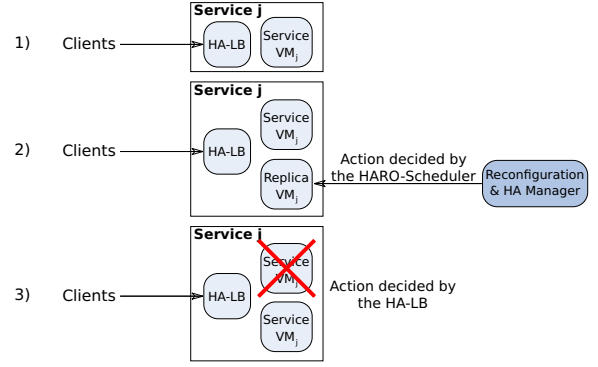

Fig. 3: Recovery architecture

We note that the rejuvenation process is based partially on the rejuvenation technique presented in [11]. It could be used to check the details of the process and its effectiveness to avoid the crash or downtime of the service during the process. The previous approach used a primary service and a hot standby service, running simultaneously. This approach consumed too many resources. Current approach is possible because the *AgingPrd* estimates the time to crash, with an acceptable error. This makes possible the replicaVM creation in advance.

Obviously, the rejuvenation process is potentially useful for any predictable software failure. The rejuvenation approach is also ready to detect sudden crashes (via heartbeat). The replica creation is triggered automatically in the first node available, without taking into account the resource optimization.

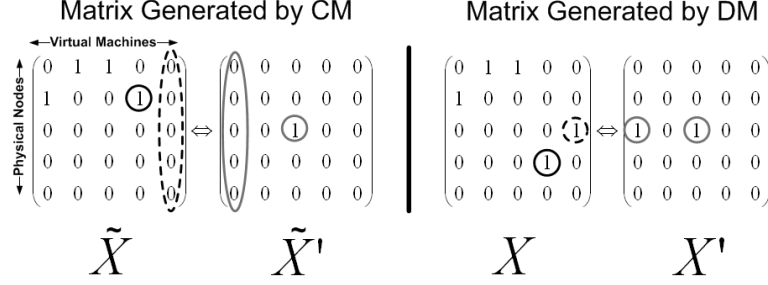## III. PROBLEM FORMULATION AND DISCUSSION

We have presented the *Decision Manager* module which has the responsibility of looking for the optimal position of VMs to accept as many services as possible. At the same time, the *DM* has to detect any faulty service and determine the best place to create its replica. This process becomes critical to guarantee the availability of the services running on the virtualized platform.

The problem to solve by the *Decision Manager* could be reduced to well-known resource assignment problem (RAP) with small variations, as follows:

"For a given set of physical nodes, and for a given set of virtual machines with a size and availability requirements; decide to which physical node each virtual machine must be allocated, such that number of virtual machines is maximized and virtual machine requirements are satisfied without exceeding physical nodes' capacity and whole platform capacity limits."

The novelty of our approach is that we integrate in the same mathematical model the decision of rejuvenate a service and according to this, calculate a maximization function to accept as many new services as possible. This approach increases the effectiveness of the maximization. The mathematical model reduces the impact of the creation of replicas (consuming resources) to apply the rejuvenation.

The problem presented could be formulated as a Constraint Satisfaction Problem (CSP). There are potentially many algorithms for tackling this kind of problems: greedy algorithms, Tabu search, genetic algorithms, and simulated annealing [14]. We use *mathematical programming* (MP) [15] because it is a common and flexible technique for modeling a large class of optimization problems. A

Fig. 2: Matrix evaluation example conducted by *RHAM*

second reason to use MP is that there are commercially and free available MP solvers, extensively tested such as IBM ILOG CPLEX [16] or GLPK [17]. We have picked the last free and open source solver in our experiments. The third reason is that MP allows us to model a large number of hard constraints, a characteristic item of RAP problems, which could be more complicated using other approaches such as genetic algorithms. However, MP requires good definition models to become useful.

We present the modeling using MP approach and its reduction to a *binary integer programming* (BIP), a subset of MP, which allows us to find the optimal solution, if it exists.

We note that from the *DM* perspective there are two type of elements: Deployed Virtual Machines and Service Containers waiting to be deployed. Deployed virtual machines are composed of ServiceVM and HA-VM and potentially a Replica VM with their respective sizes. On the other side, Service containers would be also composed of two VMs: Service VM and HA VM. However, from DM perspective becomes in a single logical VM. This logical VM has a size which is the sum of the sizes of both future VMs. Finally, we have introduced a timeout in the solving engine (GLPK) to avoid huge time consuming to achieve the optimal solution. We have selected this option because sometimes it is impossible to find it or the time needed is prohibitive.

Note that we will have to model cumulative resources, because we assume that the provider is not allowed to remove from the platform any service already deployed and under execution. However, customers could request to remove a service at any time.

A. Objective Functions

Any MP problem is composed by an Objective function and a set of constraints. The problem input is the matrices generated by the *Catcher Manager*: $\widetilde{X}$, $\widetilde{X}'$ and W. The output is composed of two matrices: X and X'. Before presenting the objective function, there are two important actions in a virtualized platform to take into account: virtual machine creation and live-migration. The creation has an impact on the rest of VMs running on the same physical node where the new VM is created. The live-migration can cause an outage (minimum) on the VM. These both actions have an impact on the performance and availability of the service. Based on that potential impact, we present two MP models: *Restrictive* and *Mixed*.

Restrictive Model (RM) penalizes both actions: creation and migration. Assuming that we have N physical nodes and M virtual

machines to be allocated in the virtualized platform; the objective function of this model integrates the penalization of both actions as follows:

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - C \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j) \quad (1)$$

where:

$$f(i,j) = \begin{cases} X_{i,j} & if\, \widetilde{X}_{i,j} = 0 \\ 0 & if\, \widetilde{X}_{i,j} = 1 \end{cases}$$

V_j represents the value/priority assigned to every VM j. The model allows assigning different priorities to VMs, guaranteeing the acceptance of VMs over others. In our experimental study, all VMs are equally important ($V_j = 1$).

To avoid that the penalization ($\sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j)$) dominates the maximization function, we introduced a penalty factor C. In our experiments, $C = \frac{1}{2 \cdot M}$. This value was defined to reduce the penalization according to the number of VMs expecting to be allocated. The penalization function $f(i,j)$ is defined as follows: If a position of the matrix $\widetilde{X}$ is zero, this means that a VM could be allocated, during optimization process, via live-migration or creation. If this happens, this position in matrix X will be 1, and we sum this value to the penalization, otherwise we do not penalize at all.

Mixed Model (MM) sits on the fence between *Restrictive Model* and no penalties. The creation is necessary to maximize the number of services running on the platform. Penalizing the creation becomes in an underutilized resources, not accepting new services in some cases. Due to this reasoning, we modified Equation 1 to penalize only the live migrations. Thus we avoid rejecting new service requests unnecessarily.

$$Maximize \sum_{i=1}^{N} \sum_{j=1}^{M} V_j \cdot X_{i,j} - C \sum_{i=1}^{N} \sum_{j=1}^{M} f(i,j) \quad (2)$$

where:

$$f(i,j) = \begin{cases} 0 & if\ \sum_{k=1}^{N} \widetilde{X}_{k,j} = 0 \\ X_{i,j} & if\ \sum_{k=1}^{N} \widetilde{X}_{k,j} = 1 \end{cases}$$

The penalization function $f(i,j)$ is performed according to if a VM could be migrated. If column j of $\widetilde{X}$ which represents VM j, is all zeros, VM j is waiting to be deployed. If there is a 1 in column

j, this means that VM is running and could be migrated. So, it only penalizes the current live-migrations.

B. Constraints

The maximization functions are not enough by themselves. It is needed to carry out a set of constraints to achieve the goals:

- Every physical node can manage a limited number of VMs, according to its resources and this limit cannot be exceeded.
- A VM cannot be removed from the platform after it was deployed.
- If a service crash is imminent, a service rejuvenation process has to be started automatically, to guarantee the availability of the service.

1) First Constraint: Due to the current virtualization technology, the number of VMs in a single node is limited by the physical resources available in the node. The model has to guarantee that the capacity (resources) limitations are not violated. We note that during a short period of time the serviceVM and replicaVM are running simultaneously. The replicaVMs are consuming extra resources as well. The constraint is defined as:

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \sum_{j=1}^{M} W_j' \cdot X_{i,j}' \leq C_i \qquad (3)$$

Where C_i is the maximum capacity of node i and W_j is the size of VM j. $X_{i,j}'$ represents a replicaVM j running in node i and W_j' is its size. This constraint guarantees the model never adds a new virtual machine to one node if there are not enough resources.

However, this constraint allows the model to consume all available resources in all nodes with new services. If the model consumes all nodes' space with new services, all resources could be *saturated* ($\sum_{j=1}^{M} W_j \cdot X_{i,j} = C_i$). Then, if a replicaVM is needed, the framework does not have enough space to conduct the rejuvenation process and the service will crash. This situation is unacceptable.

The model has to overcome the *saturation*. The framework reserves a part of the resources of the platform to guarantee that there will be enough space to create replicas at any time. Or at least, there will be enough space to create replicas in some order to avoid unplanned downtimes caused by aging. We define the space reserved in the platform to offer high availability as β. Although, it is impossible that a virtual machine uses simultaneously resources from two different nodes. So, the model allocates β_i space in every node. The β_i and the size of VMs defines the maximum number of simultaneous recovery process in the platform as follows:

$$\text{Total num. of simultaneous replicas} = NumNodes \cdot \frac{\beta_i}{W_j} \qquad (4)$$

(assuming all replicas have the same size W_j)

If no aging service is detected ($\forall j, 1 \leq j \leq M, TTC_j > TL$, where TL_j is Time Limit (TL) of service j), all physical resources from every node can be used to deploy VMs, always preserving β_i space/resources to guarantee the availability. However, if any running VM violates its own time limit threshold, some of the β_i spaces reserved have to be released to guarantee the rejuvenation process success. At the moment of releasing any of the β_is, model guarantees that this space is only used to create replicaVMs. Furthermore, when *ReplicaVM* becomes the new *ServiceVM*, β_i has to be recovered.

The constraint is focused on using β_i from the same node where the faulty serviceVM is running. Meanwhile, the rest of nodes accept new services. We define M' as the total number of running VMs

and M the total number of VMs, where $M' \leq M$, the constraint becomes as follows:

If $\forall j, 1 \leq j \leq M, TTC_j > TL_j$ then:

$$\forall i, 1 \leq i \leq N, \sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i$$

Otherwise:

$\forall j, 1 \leq j \leq M'$, with $TTC_j \leq TL_j$:
For the unique i where $X_{i,j} = 1$:

$$\forall j, M' < j \leq M, X_{i,j} = 0$$
$$\wedge$$
$$\sum_{j=1}^{M'} W_j \cdot X_{i,j} + \sum_{j=1}^{M'} W_j' \cdot X_{i,j}' \leq C_i$$

While the rest of i's:

$$\sum_{j=1}^{M} W_j \cdot X_{i,j} + \beta_i \leq C_i \qquad (5)$$

The model only rejects new services in the node i where a rejuvenation action is triggered. This approach reduces the possibility to reject new services.

2) Second Constraint: We cannot drive out any running service. This action will correspond to a crash of a service. The model guarantees that a VM already deployed on the virtualized platform is maintained. Furthermore, due to the characteristics of MP, we have to avoid explicitly the ubiquity of a VM. But a service owner can deploy as many new services as he wants. Although, a determined VM is deployed once. This constraint is modeled in Equation 6:

$$\forall j, 1 \leq j \leq M', \sum_{i=1}^{N} X_{i,j} = 1$$
$$\wedge$$
$$\forall j, M' < j \leq M, \sum_{i=1}^{N} X_{i,j} \leq 1 \qquad (6)$$

Where M' represents the VMs deployed on the infrastructure and M the total number of VMs: the running VMs and waiting services. The first part of the constraint guarantees the running VMs ($\forall j, 1 \leq j \leq M'$) have to be maintained on the environment. The second part guarantees that the waiting VMs could be accepted only once.

3) Third Constraint: The Software Rejuvenation Constraint: Finally, the model has to manage the software rejuvenation approach. When a serviceVM crash is imminent due to software aging, a replicaVM has to be created. Our model takes the responsibility to decide when a replica creation is needed and where is the best place to do it. Equation 7 models it.

$$\forall j, 1 \leq j \leq M, f(X_{i,j}') = \begin{cases} \sum_{i=1}^{N} X_{i,j}' = 1 & \text{if } TTC_j \leq TL_j \\ \sum_{i=1}^{N} X_{i,j}' = 0 & otherwise. \end{cases}$$
$$(7)$$

Where TTC_j is the time to crash of the VM j, and the TL_j (Time Limit) is the rejuvenation threshold for VM j. So, if $TTC_j \leq TL_j$ a *Replica VM* of j has to be created, otherwise not. The value of TL_j is very important to guarantee the availability of the services against software aging. We suggest that the best value of TL_j has to be a sum of the time to create replicaVM, the error introduced by the predictor, and a security margin. The security margin is added only to improve the robustness of the framework against the prediction accuracy error.

On the other hand, if $TTC_j > TL_j$ no replicaVM is needed by service j.

IV. EXPERIMENTAL SETUP

The experimental study was conducted on the test bed machines presented in Table I. All machines use a Debian GNU/Linux 5.0 as Operating System and a XEN 3.3.2 hypervisor [3] provides paravirtualized environment and all the working nodes support NFSv4 as File System in order to apply live migration between nodes. The virtualization middleware used to create/destroy virtual machines was EMOTIVE [18].

	Experiment Role	CPU	Memory Size
Client	Clients	8 core 2.83GHz	16384MB
node0	Scheduler/Node	8 core 2.83GHz	16384MB
node1	Node	8 core 2.60GHz	16384MB
node2	Node	4 core 3.0GHz	16384MB
node3	Node	4 core 2.66GHz	16384MB
db0	DB	4 core 2.66GHz	4096MB
db1	DB	4 core 3.16GHz	4096MB
db2	DB	2 core 2.40GHz	2048MB

TABLE I: Description of Testbed machines and their roles

The virtualized platform is composed of 4 nodes: from 0 to 3. The resources required by the virtual machines used in our experiments are depicted in Table II.

	Memory Size	Disk
Domain-0	4096MB	–
Service VM	2048MB	3703MB
HA VM	256MB	801MB

TABLE II: Description of Virtual Machines

Domain-0 is required and mandatory by the host operating system. So, the available memory in every physical node is the substraction of Domain-0 memory required from total physical memory available. It is important to remark that there is a *Domain-0* VM in every node where we want to apply virtualization technology.

We have used TPC-W benchmark [19] to emulate the services to be deployed on the platform. We have modified TPC-W to emulate a software aging phenomenon due to resource exhaustion. To simulate the resource consumption we have modified a servlet of the TPC-W implementation to inject memory leaks. The servlet calculates a random number between $0..N$ (in our experiments $N = 30$). This number determines how many clients have to request the servlet before to inject a memory leak. This behavior makes that the variation of memory consumption depends on the number of clients and the frequency of servlet visits. The frequency is based on the workload available in TPC-W: Browsing, Shopping and Ordering. We have conducted all of our experiments using Shopping. In this case the clients visit the Servlet modified around 20% of times. This makes that with high workload our servlet injects more memory leaks. However with low workload, the consumption is lower too. The memory leak

injected in our experiments is fixed (1MB). This configuration makes that the aging phenomena is workload dependent. This rationale has been proposed in several related works [10], [20]. Under this scenario, *AgingPrd* prediction algorithm showed its great accuracy to estimate the time to crash [10]. The framework is independent of the prediction mechanism/approach used. New prediction/estimation approaches could be integrated on the framework.

In our scenario, a *Service VM* is composed of a web application server with TPC-W. In our experiments, the databases used by TPC-W are externalized to simplify the experimental environment, due to a limitation about available physical devices. However, they could be managed by the infrastructure without problems. The software aging phenomena was injected only in the *ServiceVMs*.

V. EXPERIMENTAL EVALUATION

In this section we present the results obtained after evaluating the models proposed. First, we have analyzed the maximum service capacity of our scenario using different configurations. This analytical analysis is useful to evaluate the real resource penalty introduced by our approach to guarantee the availability.

Second, we have conducted a set of experiments to evaluate the overhead introduced by our architecture, mainly, knowing the overhead introduced by the creation of the *HA-VM* associated with the service deployed by the customer.

Third, the live migrations could cause potential micro-outages [13]. We have measured its real impact in our scenario.

Fourth and finally, in Section III we have presented two main strategies gathered in two models: *Restrictive (RM)* and *Mixed (MM)*. We compare the two models based on following metrics: maximum services accepted, number of migrations conducted, capacity to create replicaVMs and avoid platform saturation, and time needed to accept the maximum number of services.

The first metric evaluates if the models achieve the maximum value obtained analytically. The second metric evaluates the effectiveness of the penalizations and the potential risk of suffer micro outages during the process. The third metric evaluates if the models are able to avoid the crash caused by aging of a deployed service.

The last metric is relevant in order to evaluate the capacity of the framework to accept new clients. If the framework causes that service owners have to wait too much time to deploy their services, it is possible that customers move on to other providers. Furthermore, this metric evaluates the impact of the creation penalization defined in the *Restrictive Model*.

A. Analyzing the maximum capacity of our platform

Taking into account the virtual machine and the physical node features, we calculated the amount of VMs that fit in our experimental environment, presented in Table III. In order to simplify the analysis and later experimental study, the VM *size* is only computed with the memory required. But, other resources can be discussed. The *size* of the *HA-VM* is 256MB and the *ServiceVM* (also *ReplicaVM*) *size* is 2048MB (Table II). Every physical node has 16384MB of memory, but *Domain-0* needs 4096MB. Thus they have 12288MB available (see Table I). The platform is composed of 4 physical nodes, then $12288 \cdot 4 = 49152$MB total memory available. If we only deploy *ServiceVMs*: $\frac{49152}{2048} = 24$ *ServiceVMs* are available to be running simultaneously.

Using our approach (with a *HA-VM* associated to every *ServiceVM*), the maximum number of services would be 21: $\lfloor \frac{49152}{2304} \rfloor = 21$ VMs. Our approach is losing around of 12.5% of the resources to guarantee the availability of the deployed services. This penalty

is due directly to the architecture of our rejuvenation solution. These numbers are calculated without reserving any β_i to guarantee enough space to create replicas. *HA-VM* and its *ServiceVM* can run in different nodes because our platform is composed of physical nodes where the networking time becomes negligible, as well as the response time of the service.

	Maximum VMs allowed	Penalty associated
Basic Scenario	24	–
Our $\beta_i = 0$MB Scenario	21	12.5%
Our $\beta_i = 2048$MB Scenario	17	29.1%
Load Balancer Scenario	11	54.1%

TABLE III: Maximum number of Services accepted

Allocating a part of the resources of every node to guarantee the replica creation as it is described in Section III-B, the maximum number of VMs depends on the value of β and β_i. In our experiments, we fix $\beta_i = 2048$MB. This value was chosen because one *Replica VM* has this size. So, the maximum number of VMs would be 17, becoming in a penalty of 29,1%.

However, if we decided to use a typical load balanced cluster solution with a load balancer (256MB) and two *Service VMs* (2048MB) (or even such as our previous solution presented in [11]), only 11 services can be deployed. This configuration pays around of 54.1% of the infrastructure to guarantee the availability. So, our architecture by itself reduces the resources dedicated to offer high availability against traditional approaches. Paying 12,5% or 29,1% of the resources to guarantee the services availability represents a good trade-off.

B. Impact of the Framework Infrastructure

This section analyzes the overhead of our creation approach. The creation of the *HA-VM* is automatic and transparent to the service owner. The *HA-VM* and *ServiceVM* are always created in the same physical node at first time. This conduct can be discussed because if the *HA-VM* and the *ServiceVM* are created in different physical machines, it can reduce the overhead introduced by the creation process. But that approach has only sense in a cluster within the same network, because if the scenario was a Cloud (involving WANs) our approach has more sense in order to reduce the response time between the *HA-VM* and its associated *ServiceVM*.

We have used EMOTIVE virtual machine creation process, which is described in [21]. We used the *node0* as the physical node to conduct the experiment. We define the creation process from the moment where the customer requests the creation of the VM until the moment of service is ready to attend the first task/request. Table IV shows the average creation time of *HA VM*, the *Service VM* and the both simultaneous creations (Full service creation). We conducted the experiment 10 times.

	Average Time	Standard Deviation
HA VM Creation	34.6s	± 11.1s
Service VM creation	64.5s	± 21.7s
Full Service creation	128.1s	± 5.1s

TABLE IV: Virtual Machines and service creation

The creation time of the Full Service is higher (128s) than the time needed to create both VMs consecutively (99.1s). It shows clearly the important impact of creating simultaneously two virtual machines in the same node. For this reason, our approach creates in sequence both MVs. In a first view, the time needed to create the software rejuvenation components is too high. However, the creation process is conducted only when a new service is deployed first time, becoming in an acceptable trade-off if we achieve a high level of availability for the services deployed.

C. Impact of the Live-Migrations

The second experiment evaluates the real impact of the live-migrations in our scenario. Obtaining the optimal allocation of the VMs requires live-migrations to reallocate deployed VMs to free space for new ones, becoming in a critical piece of the framework. We measured the potential outage on the service when a VM is live-migrated from one node to other. We have setup a *ServiceVM*.

First, we analyzed the impact on the static content web service on Tomcat using httperf [22] and a workload of 500 new requests/sec. The experiment was running for 90 minutes. First time without migrations and second time, 12 live-migrations were performed. The average performance with and without migrations was 498.6 req/s and 500req/s respectively. The response time was 0.7ms and 0.4ms respectively. We can observe the minimum impact on response time and throughput in the case of static content web service.

After that, we conducted a new experiment using TPC-W benchmark (with dynamic content). We run 8 migrations during 60 minutes. Figure 4 presents the throughput and response time obtained and the migration marks (black lines). We observe that the migration does not impact on response time or throughput. The fluctuations observed are caused by the dynamic content and the different SQL queries times.
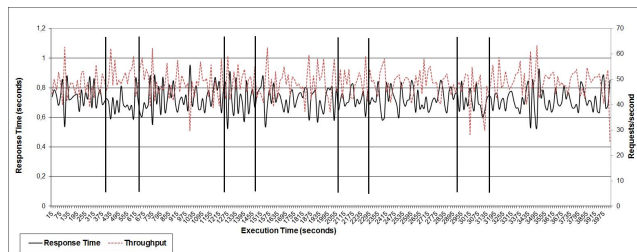

Fig. 4: TPC-W Throughput with eight live-migrations

D. Model Comparison

Finally, we evaluate and compare the two presented models: *RM* and *MM*. *HARO Scheduler* manages the four working nodes which will host the VMs.

The experiment consists of the submission of a new service every five minutes. We create 50% of services without software aging and 50% with it, interspersing them. The new creation requests are generated until the system reaches its maximum capacity, a *ServiceVM* crashes or a *ReplicaVM* could not be created. The last reason will cause a future outage in one service due to the software aging. We defined this fact as *Saturation of the platform*.

Different workloads are used in every service to increase the complexity of the software aging scenario. We find different aging trends in every faulty service, so the crash of every service will happen at different instants. We used the same trained Machine Learning model for all services in order to analyze its effectiveness to predict different aging trends, analyzing its flexibility and adaptability.

We used the same new service requests schedule for all experiments conducted. We also assign the same workload to every VM in all experiments, to allow us comparing results.

In terms of scheduling, every 60 seconds, every *AgingPdr* sends its TTC_j to the *HARO Scheduler*. Then, the *Decision Manager* runs

the model (*RM* or *MM*) to obtain the optimal allocation of VMs according to the current state.

We configured the parameters of our strategies as follows: $\beta_i = 2048MB, \forall i, 1 \leq i \leq N$, where $N = 4$ and $TL = 1000$ seconds. TL is calculated summing 900s from approximated error of *AgingPdr* prediction and 100s as a security margin.

Once the platform reaches the maximum capacity, the experiment lasts another 4000s in order to test the stability.

We have also defined three more models as a baseline comparison: *Base case*, $C = 0$ model and $\beta = 0$ model. The base case represents a *RM* model without third constraint (availability Constraint) and no penalization ($C = 0$). The *Base case* only optimizes the resources available, without taking into account aging phenomena on the services. $C = 0$ Model is the *RM* model with no penalizations. The $\beta = 0$ model is the *RM* model without reserving resources to guarantee the service availability: β and β_i equal to zero.

These three models are used to evaluate the real impact of the penalizations introduced and the availability constraint defined.

1) Maximum Services Accepted: In Table V, we can observe how the *Base case* only accepts 12 services before one of them crashes due to the software aging. This model does not conduct any migration because it was unnecessary. However, we note that in this scenario there are probabilities to achieve the maximum possible: 21 services. The $\beta = 0$ model also reaches the maximum number of services possible on the platform, 21 services, as we analyzed in the previous analytic evaluation. At the same time, models $C = 0$, *RM* and *MM* achieve the maximum number of services assuming that they preserve a part of their resources to guarantee the availability. The maximum number of services reached was 17.

The number of replicas (Table V) depends mainly on the workload and software aging aggressiveness. However, this value demonstrates that the models with our framework are able to apply the necessary recovery actions.

The number of migrations (Table V) shows clearly how the penalty introduced in *Restrictive*, *Mixed* and $\beta = 0$ Models works perfectly compared with $C = 0$ Model: 52 migrations against 11 (*RM*), 3 (*MM*) and 4 ($\beta = 0$ Model).

Model	Services	Migrations
Base case	12/21[a]	0
$\beta = 0$ model	21	4
$C = 0$ model	17	52
RM	17	11
MM	17	3

[a] It is possible in some circumstances to achieve the maximum.

TABLE V: Maximum services accepted by the models

2) Rejuvenation Analysis: Our ultimate goal is to guarantee the availability of the aging services. For this reason, we evaluate the Machine learning approach to predict aging services, creating replicas and analyzing if the platform could be saturated.

Table VI presents the number of rejuvenations conducted by the models: $\beta = 0$ model, $C = 0$ model, RM and MM. And the time needed by the models to be saturated or accept the maximum possible services.

We recall the complexity of the aging scenario with different aging trends. In this complex scenario, we used one unique trained model for all aging trends.

During all rejuvenations any request (new or on-going) failed, confirming that the evolution of the software rejuvenation maintains the results presented in [11], [12]. A more detailed analysis of the rejuvenations triggered shows an interesting result about the prediction mechanism based on Machine Learning. In the case of $\beta = 0$ model, 6 rejuvenations were conducted, with 5 false positives. This means that the framework rejuvenated non-faulty services. In the case of RM and MM, 2 false positives. In the case of $C = 0$ model no false positives found, basically because the platform was saturated early that happens. No false-negatives were found during the experiments. These results show that pessimistic estimations are better to guarantee the availability of the services. It is important to note that if some service TTC oscillates between a borderline dangerous state and non-dangerous state, current version of the models rejuvenate it. However, this is not quite dangerous because this process does not impact on the availability of the service.

Finally, we analyze the time needed by every model to create all services requested during the experiment. We also analyze the real impact of the penalization in the RM. It needs 8422 seconds to accept the same services that $C = 0$ model and MM. This shows the impact of the creating penalization, adding near of one hour more to accept the same number of services. This means that the service providers wait one hour more to deploy their services. Something undesirable. In terms of saturation, Base case model saturates after 3351 seconds of experiment started. A service crashes when only 12 services were accepted.

Of course, the maximum time to accept all service is affected by the number of rejuvenation actions conducted. The rejuvenation action in one node avoids the creation of new services in that node.

We also observe that all models with rejuvenation ($\beta = 0$, $C = 0$, RM and MM) conduct the rejuvenation on the same faulty service. RM also conducts a second rejuvenation action over a real aging service due to the execution time length.

According to results presented in Table VI and Table V, the best option is the *MM*: maximum services, minimum migrations and maximum availability.

Model	Max.Time	Saturation	Rejuvenations[a]
Base case	-	3351secs	0
$\beta = 0$ model	6958secs	6958secs	6/5[a]
$C = 0$ model	5951secs	-	1/0
RM	8422secs	-	4/2
MM	5125secs	-	3/2

[a] Total/False positives

TABLE VI: Availability achieved by the models

3) Models Behavior: Figure 5 presents the evolution over time of the number of VMs and Services managed by Mixed Model. The number of services is always going up because any service crashed during the experiment. However, the number of VMs goes down around instants 3500, 4000 and 4100 because three services where rejuvenated (3 replicas, Table V). These instants show when a primary *ServiceVM* was replaced by a *ReplicaVM*. This fact reduces the number of VMs on the platform, because during a short period of time the primary and replica are running simultaneously, and after that the faulty serviceVM is destroyed.

Figure 6a shows the number of VMs allocated in each node along the time. The figure shows the events (creating and destroying VMs) related to its instant time. This is the reason why the time scale is not homogeneous. It has a first stage, until second 3800, where the system is creating new services. The model tries to allocate in a balanced way the new services between all nodes. As model penalizes the live-migrations, it is allocating the new services using a free node until

the node is full. For example, *node2* is not used until second 2700 approx. when the *node1* is full. At this moment, no live-migrations were conducted. So, all HA-VMs are running in the same node as their serviceVMs.
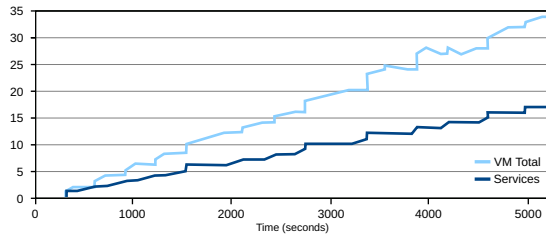

Fig. 5: Experiment execution using *MM*: Amount of Services and VMs

When all nodes are full without migrations, the model starts the VM migrations to free space to accept more services (instant 4900s). We observe the three replica creation process (at 3500, 4000 and 4100 seconds).

Figure 6b presents the behavior with $C = 0$ Model (no penalizations). The migration process starts too early when it is completely useless to free more space. The platform has enough space to allocate more services. Just recalling, $C = 0$ model follows the same event-pattern as Mixed Model in Figure 6a. We can observe how at instant 1700 *node3* creates 2 VMs (first peak) and 200s later, these two VMs are migrated to *node0*. This process is repeated twice more. This type of behavior causes a unnecessary number of migrations (52). The model is stateless, making that the model does not check if the current state of the platform is already optimal.

VI. RELATED WORK

The use of virtualization to consolidate services and improve the resource utilization is not a new idea and it has been evaluated largely in several studies [23], [24]. Furthermore, the usage of *mathematical programming* to optimize the service allocation is also not new. The MP approach has been used in several studies. Pmapper [25] uses the maximization function to improve the energy efficiency of a virtualized clusters. Furthermore, we can observe parallelism between their approach with our approach. They penalize the service migration because they assume that live-migration has an important energy cost. [26] presents a model to maximize the number of services to deploy in a virtualized data center, similar to our approach, but they do not take into account the availability as a factor on the model. [27] evaluates how to allocate every tier of a multi-tiered application to maximize the response time of those type of applications. It has also been used for minimizing the power consumption and maximizing the performance of a virtualized cluster [28].

On the other hand, we can find several studies where the virtualization is used as an elegant way to improve the dependability and availability of the Software and Hardware systems [29]. Remus [30] uses asynchronous virtual machine replication to provide high availability to server in the face of hardware failures. In [31], authors present Mercury, an on-demand virtualization for HPC clusters. Mercury offers the possibility to switch from virtualized to native mode without important penalty. This approach is quite interesting because, they achieve to maintain the important capabilities of the virtualization like live-migration or checkpointing, reducing the overhead introduced by the virtualization layer. This solution could be integrated with our solution without missing any advantage of our approach. [32] presents the use of virtualization technologies to offer

dependability in Avionics. Mainly, the work uses the virtualization technology to ensure dependability of critical avionic applications. The avionic applications have to communicate with off-board systems without the same level of validation that suffers on-board systems. The virtualization avoids the communication with off-board systems.

In [33] is presented a work similar to us. They presented a model to offer a self-reconfiguration system for HPC virtualized environments and propose a set of strategies to select the nodes to deploy the tasks to run. They take into account the performance and the reliability status of the node to be selected. They use a proactive mechanism to calculate the probability that a node could fail while the task is running. Moreover, they evaluate the current workload on the node and using all of this information, the best node is chosen. [34] is other close-related work to our approach. In this case, the authors are focused more on maximizing performance and guaranteeing availability than maximizing the platform resources.

[35] presents a very interesting approach to offer fast rejuvenation mechanism to avoid the consequences of software aging on the virtualization middleware more than virtual machines running over it. However, this approach does not take into account how to allocate the VMs in a virtualized platform to optimize the resource usage. Although, this solution could be integrated with our proposal to offer a full software rejuvenation mechanism on all layers of the virtualization stack.

In all related work analyzed, we can find several works using MP approaches to maximize the resource usage or works where virtualization plays a key role to guarantee the availability. However, to best of our knowledge there are not works integrating service maximization and software rejuvenation in the same mathematical model.

VII. CONCLUSIONS

In this paper we have presented a self-reconfigurable framework achieving transparent software rejuvenation solution. The platform accepts as many services as possible. The approach is based on a mathematical programming model plus a machine learning module.

Furthermore, we observed in our experiments that the best option is to give up a part of the resources to maintain all services up. We also observed that can reduce drastically the number of live-migrations (dangerous for availability of services in some scenarios) using a simple penalization function in the objective function in model.

Finally, we have presented the results indicating that this is a promising approach to service aging prediction in runtime, useful in real scenarios.

ACKNOWLEDGMENT

REFERENCES

[1] R. Figueiredo, P. A. Dinda, and J. Fortes, "Guest editors' introduction: Resource virtualization renaissance," *Computer*, vol. 38, no. 5, 2005.
[2] D. Menascé, "Virtualization: Concepts, applications, and performance modeling," in *CMG Conference*, vol. 1. Citeseer, 2005, p. 407.
[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *the 19th ACM Symp. on Operating systems principles*, 2003.
[4] VMWare website. [Online]. Available: http://www.vmware.com
[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *the 4th conf. on USENIX Symp. on Internet Technologies and Systems*, 2003.
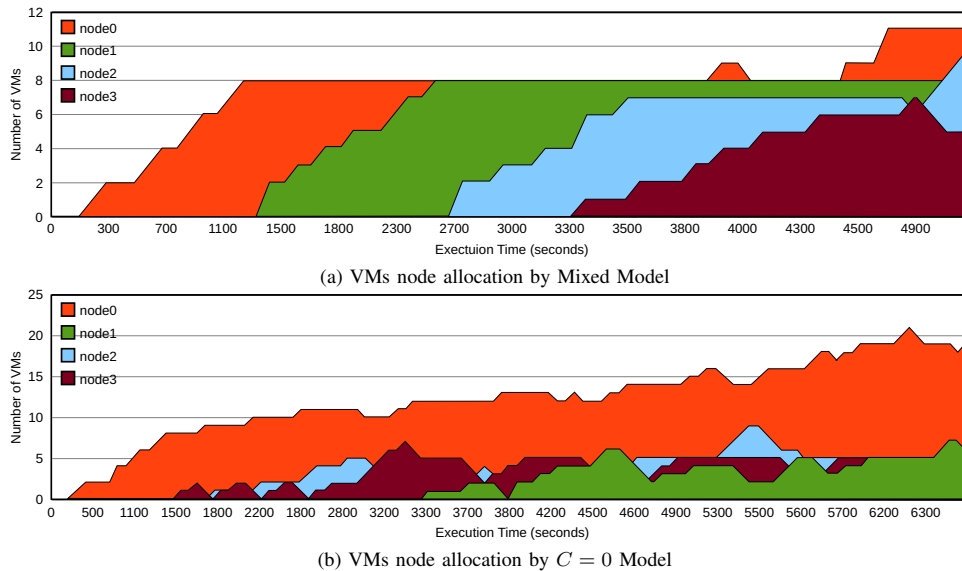
(a) VMs node allocation by Mixed Model

(b) VMs node allocation by $C = 0$ Model

Fig. 6: Comparison of VM allocation of *MM* (a) and $C = 0$ Model (b)

[6] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Parallel Data Laboratory, Carnegie Mellon University*, no. CMU-PDL-05-109, 2005.

[7] E. Marcus and H. Stern, *Blueprints for high availability*. John Wiley & Sons, Inc., 2003.

[8] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *the 15th International Symposium on Fault-Tolerant Computing (FTCS '95)*, 1995.

[9] Y. Wang and I. H. Witten, "Inducing model trees for continuous classes," in *the 9th European Conf. on Machine Learning Poster Papers*, 1997.

[10] J. Alonso, J. L. Berral, R. Gavaldà, and J. Torres, "Adaptive on-line software aging prediction based on machine learning," in *The 40th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2010.

[11] L. M. Silva, J. Alonso, and J. Torres, "Using virtualization to improve software rejuvenation," *IEEE Trans. Comput.*, vol. 58, no. 11, 2009.

[12] J. Alonso, L. Silva, A. Andrzejak, P. Silva, and J. Torres, "High-available grid services through the use of virtualized clustering," in *Procs. of the 8th IEEE/ACM Intl. Conf. on Grid Computing*, 2007.

[13] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *the 2nd ACM/USENIX Symp. on Networked Systems Design and Implementation*, 2005.

[14] D. L. Kreher and D. R. Stinson, "Combinatorial algorithms: generation, enumeration, and search," *SIGACT News*, vol. 30, no. 1, 1999.

[15] L. A. Wolsey and G. L. Nemhauser, *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.

[16] IBM ILOG CPLEX Optimizer, *CPLEX http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/*.

[17] GNU Linear Programming Kit (GLPK), *GLPK http://www.gnu.org/software/glpk*.

[18] EMOTIVE Cloud Barcelona, *EMOTIVE http://www.emotivecloud.net*.

[19] TPC-W Benchmark Java Version, *TPC-W http://www.ece.wisc.edu/ pharm/*.

[20] K. Vaidyanathan and K. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Trans. Dependable Secur. Comput.*, vol. 2, no. 2, pp. 124–137, 2005.

[21] I. Goiri, F. Julia, J. Ejarque, M. d. Palol, R. M. Badia, J. Guitart, and J. Torres, "Introducing virtual execution environments for application lifecycle management and sla-driven resource distribution within service providers," in *the 8th IEEE Intl. Symp. on Network Computing and Applications*, 2009.

[22] D. Mosberger and T. Jin, "httperf: a tool for measuring web server performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, pp. 31–37, December 1998.

[23] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. Shin, "Performance evaluation of virtualization technologies for server consolidation," *HP Laboratories Technical Report*, no. HPL-2007-59, 2007.

[24] W. Vogels, "Beyond server consolidation," *Queue*, vol. 6, no. 1, pp. 20–26, 2008.

[25] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008.

[26] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on Services Computing*, vol. 99, no. RapidPosts, 2010.

[27] K. Chaudhuri, A. Kothari, R. Swaminathan, R. Tarjan, A. Zhang, and Y. Zhou, "Server Allocation Problem for Multi-Tiered Applications," *Algorithmica*, 2007.

[28] V. Petrucci, O. Loques, and D. Mossé, "Dynamic optimization of power and performance for virtualized server clusters," in *the 25th ACM Symposium on Applied Computing*. ACM, 2010, pp. 263–264.

[29] B. Jansen, H. Ramasamy, M. Schunter, and A. Tanner, "Architecting dependable and secure systems using virtualization," *Architecting Dependable Systems V*, 2008.

[30] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *the 5th USENIX Symp. on Networked Systems Design and Implementation*, 2008.

[31] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Mercury: Combining performance with dependability using self-virtualization," in *the 2007 Intl. Conf. on Parallel Processing*, 2007.

[32] Y. Laarouchi, Y. Deswarte, D. Powell, J. Arlat, and E. De Nadai, "Enhancing dependability in avionics using virtualization," in *the 1st EuroSys WS. on Virtualization Technology for Dependable Systems*, 2009.

[33] S. Fu, "Failure-aware construction and reconfiguration of distributed virtual machines for high availability computing," in *the 2009 9th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, 2009.

[34] M. Hiltunen, K. Joshi, R. Schlichting, G. Jung, and C. Pu, "Performance and Availability Aware Regeneration For Cloud Based Multitier Applications," in *The 40th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, DSN 2010*, 2010.

[35] K. Kourai and S. Chiba, "A fast rejuvenation technique for server consolidation with virtual machines," in *The 37th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2007.