Dynamic Consolidation of Highly Available Web Applications

Fabien Hermenier, Julia Lawall, Jean-Marc Menaud, Gilles Muller

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic Consolidation of Highly Available Web Applications

Fabien Hermenier — Julia Lawall — Jean-Marc Menaud — Gilles Muller

N° 7545

Février 2011

Distributed Systems and Services

ISSN 0249-6399 ISRN INRIA/RR--7545--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Dynamic Consolidation of Highly Available Web Applications

Fabien Hermenier* , Julia Lawall† , Jean-Marc Menaud* , Gilles Muller‡

Theme : Distributed Systems and Services
Équipe-Projet ascola

Rapport de recherche n° 7545 — Février 2011 — 26 pages

Abstract: Datacenters provide an economical and practical solution for hosting large scale n-tier Web applications. When scalability and high availability are required, each tier can be implemented as multiple replicas, which can absorb extra load and avoid a single point of failure. Realizing these benefits in practice, however, requires that replicas be assigned to datacenter nodes according to certain placement constraints. To provide the required quality of service to all of the hosted applications, the datacenter must consider of all of their specific constraints. When the constraints are not satisfied, the datacenter must quickly adjust the mappings of applications to nodes, taking all of the applications' constraints into account.

This paper presents Plasma, an approach for hosting highly available Web applications, based on dynamic consolidation of virtual machines and placement constraint descriptions. The placement constraint descriptions allow the datacenter administrator to describe the datacenter infrastructure and each application administrator to describe his requirements on the VM placement. Based on the descriptions, Plasma continuously optimizes the placement of the VMs in order to provide the required quality of service. Experiments on simulated configurations show that the Plasma reconfiguration algorithm is able to manage a datacenter with up to 2000 nodes running 4000 VMs with 800 placement constraints. Real experiments on a small cluster of 8 working nodes running 3 instances of the RUBiS benchmarks with a total of 21 VMs show that continuous consolidation is able to reach 85% of the load of a 21 working nodes cluster.

Key-words: VM placement, cloud computing, high-availability, dynamic consolidation, datacenter

* ASCOLA Mines de Nantes - INRIA, LINA – firstname.lastname@mines-nantes.fr
† DIKU, University of Copenhagen – julia@diku.dk
‡ INRIA/LIP6-Regal – gilles.muller@lip6.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Consolidation dynamique d'applications Web haute disponibilité

Résumé : Externaliser l'hébergement d'une application Web n-tiers virtualisée dans un centre de données est une solution économiquement viable. Lorsque l'administrateur de l'application considère les problèmes de haute disponibilité tels que le passage à l'échelle et de tolérance aux pannes, chaque machine virtuelle (VM) embarquant un tiers est répliquée plusieurs fois pour absorber la charge et éviter les points de défaillance. Dans la pratique, ces VM doivent être placées selon des contraintes de placement précises. Pour fournir une qualité de service à toutes les applications hébergées, l'administrateur du centre de données doit considérer toutes leurs contraintes. Lorsque des contraintes de placement ne sont plus satisfaites, les VM alors doivent être ré-agencées au plus vite pour retrouver un placement viable. Ce travail est complexe dans un environnement consolidé où chaque nœud peut héberger plusieurs VM.

Cet article présente Plasma, un système autonome pour héberger les VM des applications Web haute-disponibilité dans un centre de données utilisant la consolidation dynamique. Par l'intermédiaire de scripts de configuration, les administrateurs des applications décrivent les contraintes de placement de leur VM tandis que l'administrateur système décrit l'infrastructure du centre de données. Grâce à ces descriptions, Plasma optimise en continu le placement des VM pour fournir la qualité de service attendue. Une évaluation avec des données simulées montre que l'algorithme de reconfiguration de Plasma permet de superviser 2000 nœuds hébergeant 4000 VM selon 800 contraintes de placement. Une évaluation sur une grappe de 8 nœuds exécutant 3 instances de l'application RUBiS sur 21 VM montre que la consolidation fournit par Plasma atteint 85% des performances d'une grappe de 21 nœuds.

Mots-clés : placement, informatique en nuage, haute disponibilité, consolidation dynamique, centre de données

Contents

1 Introduction

Most modern Web applications, such as Facebook, Twitter, or eBay, are now structured as n-tier services, comprising, for example, a load balancer, an http server, a specific business engine, and a database. Such applications must be both highly available (HA) and scalable, both of which can potentially be achieved by running the application in a datacenter and replicating its tiers. In practice, however, simply replicating tiers is not sufficient to ensure high availability and scalability. Instead, to ensure high availability, it is essential that the replicas of a given tier are assigned to nodes in such as way as that there is no single point of failure. Furthermore, to ensure scalability, replicas of stateful tiers should be placed on nodes that provide an acceptable latency, to allow the use of consistency protocols without incurring excessive overhead. These requirements should be expressed in a service contract, which is submitted to the datacenter along with the application.

Given a service contract, the challenge for the datacenter administrator is to ensure that the placement constraints will be satisfied during the whole lifetime of the application, despite the potential changes in resource availability, due to the actions of other applications and the need for maintenance of some nodes. One solution would be to place each replica on a separate machine chosen according to the HA constraints. However, since in practice the typical client load is much lower than the maximum expected, most replicas would have a low level of activity, and thus this solution would lead to a waste of computer resources and energy.

A common solution to reduce resource usage is to run multiple application tier replicas on a single node, consolidating replicas with a low activity together, thus increasing the hosting capacity of a datacenter [24]. By running tier replicas in virtual machines (VMs), it is possible to use live migration [9] to place several tiers on a single machine when their load is low, and to migrate them to different machines when their load increases [19]. In this setting, a *consolidation manager* decides when to reconfigure the datacenter and which replicas have to migrate to what nodes. Designing such a consolidation manager is challenging because it has to satisfy both the goals of the application administrators and of the datacenter administrator. Most previous dynamic consolidation systems [6, 12, 16, 19, 23, 26, 28] optimize the placement of the VMs according to their resource usage, but do not consider the application placement constraints that are required to achieve both HA and scalability. VMWare DRS [25] does allow datacenter administrators to spread VMs on distinct nodes, but does not allow the specification of any other kinds of constraints.

In this paper, we propose a dynamic consolidation manager, Plasma, that can be configured to take into account not only resource constraints but also the placement constraints of HA applications. Configuration is made through scripts that allows the datacenter administrator to describe the datacenter infrastructure and each application administrator to describe the placement constraints of the application's VMs. Plasma is built around a core reconfiguration algorithm that can be dynamically customized by the configuration scripts. Overall, our approach provides efficient dynamic consolidation while 1) guaranteeing to the application administrator that placement requirements will be satisfied and 2) relieving the datacenter administrator of the burden of considering the constraints of the applications when performing maintenance.

Our main results are

- An extensible reconfiguration algorithm which only considers an estimation of the misplaced VMs when a reconfiguration is required. This approach makes Plasma scalable to datacenter of thousands of nodes.
- A deployment on a real cluster of 12 nodes. An experiment running 3 instances of the RUBiS benchmarks with 21 VMs show that continuous consolidation can react rapidly to overload situations and can reach 85% of the load of a 21 working node cluster.
- Experiments on simulated data show that our implementation of the reconfiguration algorithm scales well up to a datacenter of 2000 nodes, 4000 VMs, and 800 placement constraints and solves such problems in less than 2 minutes.

The rest of this paper is organized as follows. Section 2 describes the global design of our system. Section 3 describes the implementation of our reconfiguration algorithm. Section 4 evaluates our prototype on a cluster. In Section 5, we discuss the flexibility of our reconfiguration algorithm. Finally, Section 6 describes related work, and Section 7 presents our conclusions and future work.

2 Plasma Design

The goal of the Plasma consolidation manager is to choose an acceptable placement for the application tier replicas, taking into account the application constraints. Plasma relies on configuration scripts for describing the infrastructure of a datacenter and the placement constraints of application tier replicas. In this section, we first introduce the configuration scripts, and then present the architecture of the consolidation manager.

2.1 Configuration scripts

Configuration scripts permit the datacenter administrator and the application administrators to each write script describing their view of a datacenter and a HA-application, respectively. Our design goals are (1) to allow a datacenter administrator to manage the datacenter's nodes without any knowledge of the placement constraints specified by the hosted applications, (2) to allow an application administrator to express constraints on the placement of the VMs that run the application tier replicas, without detailed knowledge of the infrastructure and without knowledge of the other hosted applications.

A configuration script declares sets of nodes and sets of VMs. Four kinds of constraints can be used to express restrictions on the placement of VMs: `ban` and `fence` are to be used by the datacenter administrator to specify constraints induced by administrative tasks, while `spread` and `latency` are to be used by application administrators to specify the constraints on the relative placement of an application's VMs.

Describing a datacenter The datacenter administrator must describe the available nodes, their roles, and the connections between them. A virtualized

datacenter is composed of a collection of working nodes and service nodes. Working nodes run application VMs using a Virtual Machines Monitor (VMM) such as Xen [3]. Service nodes run the services that manage the datacenter (monitoring system, resource manager, file servers, *etc.*). All nodes are physically stacked into racks and interconnected through a hierarchical network, as illustrated in Figure 1. This physical organization implies that there may be non-uniform latency between them.

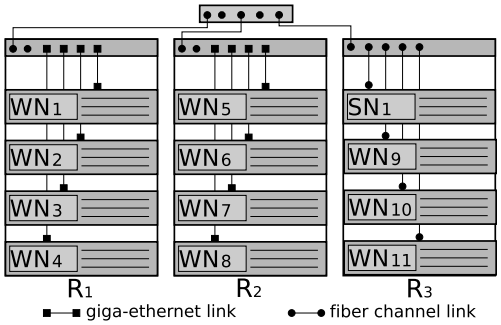


Figure 1: An example datacenter. Racks R_1 and R_2 are composed of four working nodes (WNs) each. Rack R_3 contains three working nodes, and a service node (SN). Nodes in R_1 and R_2 are connected through giga-ethernet links while nodes in R_3 are connected through fiber channel links.

The description of a datacenter provided by a datacenter administrator is illustrated in lines 1-6 of the Plasma script shown in Listing 1. Lines 1 to 3 define the variables `$R1`, `$R2`, and `$R3` as the list of nodes found in racks R_1, R_2, and R_3, respectively. Lines 7 and 8 define the available latency classes. A latency class is defined as a set of disjoint groups of nodes. Here, the class `$small` is composed of the groups of nodes connected through a fiber channel links and the class `$medium` is composed of the groups of nodes that can be reached with only one network hop.

```
1 //Definition in extension
2 $R1 = {WN1, WN2, WN3, WN4};
3 //Definition using a range of nodes
4 $R2 = WN[5..8];
5 //Definition using an union of sets
6 $R3 = WN[9..11] + {SN1};
7 $small = {$R3};
8 $medium = {$R1, $R2, $R3};
9
10 ban($ALL_VMS, {SN1});
11 ban($ALL_VMS, {WN5});
12 fence($A1, $R2 + $R3);
```

Listing 1: Description of the datacenter depicted in Figure 1. Node `WM5` is banned due to maintenance. Normally such a constraint would be provided separately, when maintenance is required, and then retracted when the maintenance is completed.

The VMs can be referenced either globally by using the variable `$ALL_VMS`, or at the level of a specific application by using the application name. Restrictions to forbid the use of a node can be expressed using a `ban` constraint, as illustrated on lines 10 and 11. This constraint is useful for service nodes, which should never host VMs, and to isolate working nodes during maintenance. Finally, it is also possible to limit some VMs to a specific set of nodes using the `fence` constraint, as illustrated in line 12.

Describing an application The application administrator must describe the VMs used by each application tier and the constraints on their placement that must be satisfied to achieve the desired degree of scalability and availability. Figure 2 illustrates a typical 3-tier Web application. The Apache services in the tier T_1 and the Tomcat services in the tier T_2 are stateless: all the handled requests are independent transactions and no synchronization of their state is needed. On the other hand, tier T_3 runs a replicated MySQL database, which is stateful: transactions that modify the datas must be propagated from one VM hosting a replica of T_3 to the others to maintain a globally consistent state. In order to provide high availability, VMs belonging to the same tier must be run on distinct nodes. In order to ensure scalability, VMs belonging to the stateful tier must be hosted on nodes with a network latency adapted to the consistency protocol that is used to synchronize replicas.

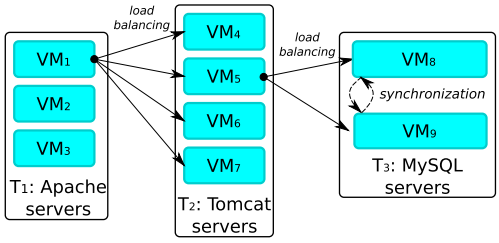


Figure 2: A 3-tier Web application.

Listing 2 presents the Plasma script for describing the constraints of the HA application shown in Figure 2. The application administrator first describes the structure of the application. Lines 1 to 3 define the variables `$T1`, `$T2`, and `$T3`, storing the VMs associated with each tier, respectively. VMs may either be listed explicitly, as in the definition of `$T1`, or defined in terms of a range, as in the definitions of `$T2` and `$T3`. The application administrator then describes any constraints on the placement of the application's VMs. In lines 4 to 6, to ensure high availability for the tiers `$T1` to `$T3`, the constraint `spread` specifies that the VMs of each tier should be hosted on distinct nodes at all times, including during the reconfiguration process. Finally, in line 7, to ensure scalability for the stateful tier `$T3`, a `latency` constraint specifies that the VMs of this tier should be placed on nodes that belong to the latency class `$medium`.

2.2 Architecture of the Plasma Consolidation Manager

The role of the Plasma Consolidation Manager is to maintain the datacenter in a configuration, i.e. an assignment of VMs to nodes, that is (i) *viable*, in that

```
1 $T1 = {VM1, VM2, VM3};
2 $T2 = VM[4..7];
3 $T3 = VM[8..9];
4 spread($T1);
5 spread($T2);
6 spread($T3);
7 latency($T3, $medium);
```

Listing 2: Description of the HA application `$A1` depicted in Figure 2 for the datacenter described in Listing 1.

all the running VMs have access to sufficient resources, and (ii) *consistent* with all the constraints specified by the datacenter administrator and the application administrators. The Plasma consolidation manager runs on a service node and initiates the reassignment of VMs to nodes when it detects that the current configuration is not viable. It consists of four modules (see Figure 3) that are executed iteratively within an infinite control loop.

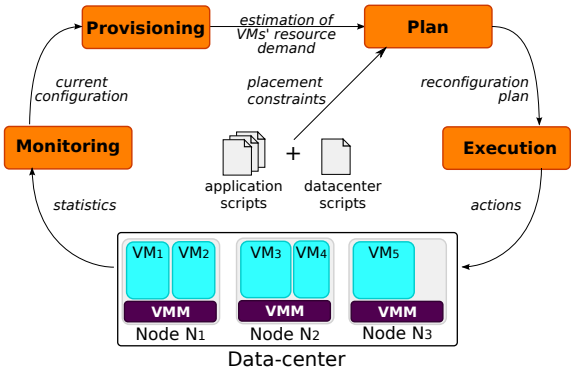


Figure 3: Plasma control loop.

The monitoring module The monitoring module retrieves the current state of each node and each running VM using the distributed monitoring system Ganglia [15]. One set of sensors, running on the top of the VMs, retrieves information about the VMs' current resource consumption while another set of sensors, running on the top of the VMMs, retrieves information about the VMMs' resource capacity and the names of the VMs they host. The computing capacity of a node and the CPU consumption of a VM are expressed using a unit, called uCPU that follows the principles of Amazon EC2 instances [1]. It provides a consistent characterization of CPU capacity that is not related to the underlying hardware.[1] All of these sensors regularly send statistics about the current state of the monitored element to a collector that is connected to the consolidation manager. When the collector does not receive statistics about a node or a VM for 20 seconds, it considers this element as offline.

[1]Estimating the uCPU capacity of a node is out of the scope of this paper.

The provisioning module The provisioning module continuously estimates the uCPU and memory requirements of each VM based on the information collected by the monitoring module. The resource usage of the replicas may change over the time, depending of the time of day, the number of clients or the type of the requests executed by the application. The provisioning module thus predicts the resource usage of a VM based on the recent changes in its consumption [27].

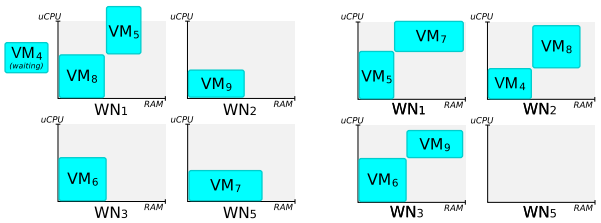
The plan module The plan module uses the configuration scripts provided by the datacenter administrator and the application administrators, the resource demand estimation provided by the provisioning module and the state of all the VMs to determined the viability of the current configuration. If all the VMs have access to the required uCPU and memory resources, and all the placement constraints are satisfied, then the configuration is viable and the consolidation manager restarts the control loop. If the current configuration is no longer viable, the plan module computes a new placement of the VMs that represents a viable configuration. Based on this information, it then computes a *reconfiguration plan*, consisting of a serie of migrations and launch actions that will relocate the VMs from their current nodes to the nodes indicated by the computed placement. The execution of the actions is scheduled to ensure their feasibility using an estimated model of their duration provided by the datacenter administrator.

The execution module The execution module applies a reconfiguration plan by performing all of the associated actions. As some action may depend on the completion of the other actions, the scheduling plan produced by the plan module is adapted to prevent the failure of the reconfiguration process if the actual duration of an action exceeds the estimated value.

2.3 Reconfiguration Example

Given the datacenter described in Listing 1 and the application defined in Listing 2, Figure 4(a) depicts a non-viable VM configuration. In this case, (i) the uCPU demand of `VM5` is not satisfied because `WN1` does not provide sufficient uCPU resources for both `VM5` and `VM8`, (ii) the node `WN5` hosts `VM7` while the use of this node has been banned by the administrator for maintenance.

The plan module computes first a new viable configuration that satisfies the resource demands of the VMs and all of the placement constraints. To reach the new configuration shown in Figure 4(b), 4 actions must be performed: `VM4` must be launched on `WN2`, `VM9` must be migrated to `WN3`, `VM8` must be migrated to `WN2`, and `VM7` must be migrated to `WN1`. In order to ensure the feasibility of the reconfiguration process, the migration of `VM7` to `WN1` cannot be performed until `VM8` is migrated to `WN2` because `WN1` does not initially have sufficient free resources to accommodate it. In addition, migrating `VM8` to `N2` before migrating `VM9` on `WN3` breaks the `spread` constraint on the tier `$T3`. The plan module then computes a schedule for executing the actions, resulting in the reconfiguration plan shown in Table 1.



(a) non-viable configuration (b) viable configuration

Figure 4: Partial sample configurations for the application described in Listing 2, running on the datacenter described in Listing 1. Each graph denotes the uCPU (y-axis) and memory (x-axis) capacity of a node. Each box denotes the estimated resource demand of a VM. VM_4 is waiting to be launched while other VMs are running.

Start	End	Action
0'0	0'5	`launch(VM4)`
0'0	0'10	`migrate(VM9)`
0'10	0'20	`migrate(VM8)`
0'20	0'30	`migrate(VM7)`

Table 1: Reconfiguration plan to reach the viable configuration in Figure 4(b) from the non-viable configuration in Figure 4(a). The launch and migration durations for a VM are estimated to be 5 and 10 seconds, respectively.

3 Implementing the Plan Module

The plan module relies on a *core reconfiguration algorithm* that takes into account the memory and CPU demands of the VMs provided by the provisioning module. The implementation is based on constraint programming [18], which allows it to be easily extended with placement constraints provided by the datacenter administrator and the application administrators. The plan module analyzes new Plasma descriptions as they are provided. It then updates the core reconfiguration algorithm according to the new constraints.

We first introduce constraint programming, then we describe the implementation of the core reconfiguration algorithm, and the implementation of the placement constraints.

3.1 Constraint Programming

Constraint Programming (CP) is a complete approach to model and solve combinatorial problems. CP can determine a globally optimal solution, if one exists, by using a pseudo exhaustive search based on depth-first search. The idea of CP is to model a problem by stating constraints (logical relations) that must be satisfied by the solution. A *Constraint Satisfaction Problem* (CSP) is defined as a set of constants, describing the current state, a set of variables, for which an assignment is to be determined by the constraint solver, a set of domains representing the set of possible values for each variable and a set of independent

constraints that represent required relations between the values of the variables. A solver computes a *solution* for a CSP that assigns each variable to a value that simultaneously satisfies the constraints.

The algorithm to solve a CSP is generic and is independent of the constraints composing the problem. In our case, this allows a deterministic specialization of the core reconfiguration algorithm and a deterministic solving process even in presence of constraints that focus on the same variables. To make the implementation of a CSP as scalable as possible, the challenges are to model the problem using the most appropriate basic constraints and to implement domain-specific heuristics to guide the solver efficiently to a solution. The Plasma consolidation manager is designed in terms of a number of standard constraints [5] that are supported by a variety of constraint solvers (ILog[2], SICSTUS[3],...). Concretely, the Plasma consolidation manager is written in Java and uses the constraint solver Choco [8] which provides an implementation for the constraints composing our model. The Plasma plan module that implements these constraints amounts to around 5500 lines of Java code.

3.2 The Core Reconfiguration Algorithm

Computing a viable configuration requires choosing a hosting node for each VM that satisfies its resource requirements, and planning the actions that will convert the current configuration to the chosen one. We refer to this CSP as the Reconfiguration Problem (RP). Each time a reconfiguration is required, the plan module uses the current configuration, the resource demand estimation provided by the provisioning module and the state of the VMs to generate a RP using the Choco API. It then inserts the placement constraints found in the Plasma descriptions into this RP. The resulting specialized RP is then solved to compute a reconfiguration plan.

The RP is defined in terms of a set of nodes $\mathcal{N}$ and a set of virtual machines $\mathcal{V}$. Each node $n_j \in \mathcal{N}$ is denoted by its memory capacity n_j^{mem} and its uCPU capacity n_j^{cpu}. During a reconfiguration, a running VM may stay on the same node or be migrated to another node. We refer to the latter as a *migration action*. A waiting VM may also be started on some node. We refer to this as a *launch action*. A *slice* is a finite period during a reconfiguration process where a VM is running on a node and uses a part of the node's resources. Slices are used within the RP to represent the resource consumption of the VMs throughout the entire duration of the reconfiguration process. We distinguish between the *consuming* slice and the *demanding* slice.

A consuming slice (*c-slice*) $c_i \in \mathcal{C}$ is a period where a VM v_i is running on some node at the beginning of the reconfiguration process before migration. The constant $c_i^h = j$ indicates that the c-slice of the VM v_i is running on the node n_j, i.e. v_i is running on n_j. Until the end of the slice, at the moment c_i^{ed}, the VM is considered to use a constant amount of uCPU c_i^{cpu} and memory c_i^{mem} resource equal to its current consumption.

A demanding slice (*d-slice*) $d_i \in \mathcal{D}$ is a period where a VM v_i is running on some node at the end of the reconfiguration process. The variable $d_i^h = j$ indicates that the d-slice of the VM v_i is hosted on the node n_j. The d-slice d_i

[2]http://www-01.ibm.com/software/websphere/ilog-migration/
[3]http://www.sics.se/isl/sicstuswww/site/index.html

starts at the moment d_i^{st} and ends at the end of the reconfiguration process, at the moment d_i^{ed}. During the whole duration of a d-slice, the VM is considered to use a constant amount of uCPU resources d_i^{cpu} and memory resources d_i^{mem} equal to the demand computed by the provisioning module.

Modeling the actions The assignment of the VMs to nodes may lead to the execution of several actions $\mathcal{A}$. The variables a_i^{st} and a_i^{ed} denote respectively the moments when an action a_i starts and ends. The duration of an action can be estimated and modeled from experiments [11]. The datacenter administrator provides, using two cost functions, a theoretical estimation of the cost of migration and launch actions, in terms of the amount of allocated memory and the uCPU consumption of the involved VM.

The assignment of a waiting VM to a node is modeled using a d-slice and will result in a launch action. Figure 5 uses a Gantt diagram to illustrate a launch action for `VM1` on node `N2`. When the action starts, the VMM allocates the memory for `VM1` and boots the guest OS. Once the guest OS is booted, the application starts and causes the VM to consume uCPU resources. Note that the d-slice continues to the end of the complete reconfiguration process, and thus may continue beyond the end of the estimated duration e_i of the migration action.

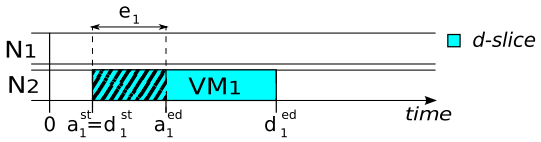


Figure 5: A placement of the waiting VM `VM1` that will result in a launch action. The action will start at the beginning of the d-slice. The hatched section indicates the duration of the action.

The activity of a running VM is modeled using a c-slice on the node on which it is running and a d-slice on the node that will run it at the end of the reconfiguration process. If the d-slice and the c-slice are not placed on the same node, there will be a migration action. The delay between the beginning of the d-slice and the end of the c-slice is then equal to the estimated duration e_i of this action. Figure 6 illustrates the migration of `VM3` from `N2` to `N1`. When the reconfiguration action starts, the VMM on `N1` allocates memory for `VM3` and starts copying memory pages, while `VM3` is running on `N2`. When the reconfiguration action terminates, the VMM on `N2` liberates resources, `VM3` is enabled on `N1` and it begins consuming uCPU resources.

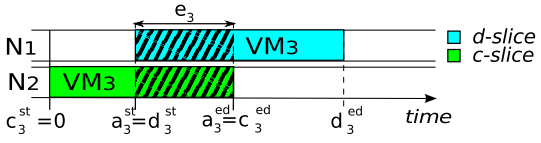


Figure 6: A re-placement of the running VM `VM3` that will result in a migration action. The action will start at the beginning of the d-slice and terminate at the end of the c-slice.

Solving a RP consists of computing a value for each d_i^h and a_i^{st}. The chosen values for these variables then indicate how to create a reconfiguration plan that contains all the actions to perform and a moment for each action to start that ensures its feasibility with regards to its theoretical duration.

VM placement To run the VMs at peak level, a node must not host slices with a total uCPU or memory consumption greater than its capacity. The final configuration is determined by the placement of the d-slices. To place the d-slices with regard to their resource demand, we use the *bin-packing* constraint [20]. As this constraint can only account for one resource, we use two instances, one for the uCPU resource and another for the memory resource.

Scheduling the actions A node may be the source for outgoing migrations, and the destination for incoming migrations and launch actions. To ensure the feasibility of the reconfiguration process, incoming actions can only be executed once there is a sufficient amount of free uCPU and memory resources on the hosting node. This implies that it may be necessary to execute some outgoing actions on a node prior to some incoming actions, to free the required resources. To plan the execution of the actions, we use a custom constraint, inspired by the standard *cumulatives* constraint [4], that computes their start and finish moments so that each incoming action is delayed just enough to wait for the required uCPU and memory resources to be freed by the termination of the outgoing actions.

Dependencies between several migrations may be cyclic. A common solution is to use an additional bypass migration on a temporary pivot node to break the cycle [2, 10, 12]. The selected pivot node hosting the VM must then satisfy all the constraints. Our model allows at most one migration per VM and selects both the placement of the VMs and the scheduling of the actions in a single reconfiguration problem. The solver may try to place a VM on a node that will lead to a cycle, but the scheduling constraint will invalidate this placement, causing the solver to directly place the VM on what would be a pivot node. Solutions of a RP are thus guaranteed to not have cyclic dependencies between the migrations while not having to use bypass migrations.

Evaluating a solution Performing a reconfiguration takes a non negligible time, during which the performance of the running applications is impacted. Executing an action consumes resources on the involved nodes and thus temporarily decreases their performance, if they do not have spare capacity. In addition, the duration of a migration increases with the memory usage of the migrated VM. Finally, delaying a migration that would reduce the load of an overloaded node maintains a non-viable configuration that reduces the performance of the application.

Given a notion of cost, a constraint solver can compare possible solutions, and only return the one that has the lowest cost. Given that both migrating a VM and delaying this migration have an impact on the overall performance, we construct a notion of cost that takes both of these durations into account. Specifically, the cost K of a reconfiguration corresponds to the sum of the elapsed time between the moment when the reconfiguration starts and the moment when each reconfiguration action has completed. This measure takes into account the

number of actions, their execution time, and their delay. K divided by the number of launch and migration actions gives an approximation of the duration of the reconfiguration process.

Optimizing the solving process Computing a solution for the RP may be time consuming for large datacenters as selecting a host for each VM and planning the actions is an NP-hard problem.

Our first approach to reduce the solving time of the RP is an heuristic that restricts the number of running VMs to consider in the RP to a minimum. Once the RP is specialized by the placement constraints, each checks the viability of the current configuration and returns in case of failure a set of *misplaced* VMs to try to replace, that is expected to be sufficient to compute a solution. We refer to these VMs as the *candidate* VMs. The placement variables of the other running VMs will be assigned to their current location before starting the solving process. The resulting RP is then a subproblem of the original RP, with fewer d-slices to place and schedule. As the scope of each selection heuristic is however limited to a single constraint, the set of candidate VMs may be too restrictive as there is no proof it will be good enough to make the reduced RP having a solution. A misplaced VMs selection heuristic should then returns a set of VMs bigger than a supposed minimal one.

A second solution to improve the solving process is a heuristic to guide efficiently the solver to a solution. The solver first replaces the VMs that are hosted on nodes than cannot handle their resource demand, as some of them will necessarily be migrated. Then the solver focuses on the other VMs. To reduce as much as possible the cost of the reconfiguration plan, the solver tries to place the running VMs in their current location. Finally, it tries to start the d-slices as early as possible.

3.3 Implementing Plasma Placement Constraints

The Plasma placement constraints are modeled using the variables of the RP and various standard constraints, which are provided by the Choco library. We now present their implementation.

spread The Plasma `spread` constraint ensures that VMs are never hosted on the same node at the same time, even during reconfiguration, avoiding the creation of a Single Point Of Failure (SPOF) during execution. To force the VMs to be hosted on distinct nodes at the end of the reconfiguration process, the implementation of `spread` uses a *allDifferent* [22] constraint, which forces the placement variables associated with the different d-slices to each take on distinct values. In addition, to ensure that the specified VMs never overlap on a same node during the reconfiguration process, *implies* constraints are used to delay the arrival of a d-slice on a node until the c-slices of the other involved VMs have terminated on this node. As an example, Figure 7 depicts a RP constrained by `spread({VM1,VM2})`. The solver has to migrate both `VM1` and `VM2` to compute a solution. It also has to delay the migration of `VM1` to ensure that `VM1` will not be hosted on `N2` until `VM2` is fully migrated to `N3`.

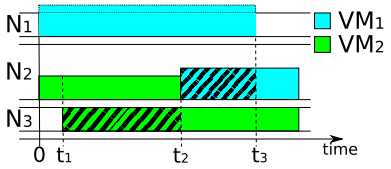


Figure 7: Using the constraint `spread({VM1,VM2})`, `VM1` and `VM2` are never hosted on a same node at the same moment to provide a full protection against SPOF.

latency The Plasma `latency` constraint forces a set of VMs to be hosted on a single group of nodes that belong to the latency class specified in parameter. A latency class $C = \{N_0, \ldots, N_a, \ldots, N_y\}$ denotes the different groups of nodes N_a that compose it. Each involved node n_j belongs to only one group, referred to by the constant $g_j \in [0, y]$. The variable $H_x \in [0, y]$ denotes the group of nodes that will host the VMs in V_x. To connect the variable H_x to the d-slices of the VMs, we use the constraint *element*. This constraint holds if $d_i^h = b \Rightarrow H_x = g_b$, which indicates that if a d-slice of a VM in V_x is hosted on a node, then all the VMs are hosted on the group of nodes that the node belong to.

fence The Plasma `fence` constraint forces a set of VMs to be hosted on a single group of specified nodes. The constraint is implemented using a domain restriction. In our context, all the nodes that are not specified as the second argument are removed from the domain of each d-slice placement variable.

ban The Plasma `ban` constraint prevents a set of VMs from being hosted on a given set of nodes. It is thus the opposite of the `fence` constraint. The implementation of `ban` uses also a domain restriction. In this context, the nodes specified in parameters are removed from the domain of each d-slice placement variable.

Overall, the translation of the placement constraints to Java is straightforward and concise. The complete Java implementation of all the constraints requires only 160 lines of code and `latency`, the most complex one, requires only 60.

4 Evaluation of Plasma

The goal of the Plasma consolidation manager is to consolidate VMs while ensuring HA and scalability requirements. However, the RP is a NP-Hard problem and thus has a potentially high computation cost for large scale clusters. We first demonstrate that dynamically consolidating VMs is interesting in practice, by evaluating Plasma on a small cluster of 12 nodes using the RUBiS benchmark [7] and demonstrate that dynamically consolidating VMs is interesting in practice. We then investigate the scalability of the Plasma reconfiguration algorithm and the impact of the placement constraints by simulating VM reconfigurations for a datacenters of up to 2000 nodes.

4.1 Evaluating the benefit of Plasma using RUBiS

RUBiS [7] is an auction site prototype modeled after *eBay.com*. It provides a 3-tier Web application and a benchmark to evaluate its performance. A single instance of the benchmark launches a given number of clients which then send requests as fast as possible. The result of the benchmark is the average number of requests per second that were successfully handled.

The experimental cluster is composed of 8 working nodes (`WN1` to `WN8`) and 4 service nodes connected through a Gigabit network. All the nodes have a 2.1 GHz Intel Core 2 Duo and 4 GB of RAM. Each working node runs Xen 3.4.2 with a Linux-2.6.32 kernel and provides a capacity of 2.1 uCPU and 3.5 GB RAM for the VMs. Three of the service nodes export the RUBiS VM images. We also use these nodes to run the benchmarks. The fourth service node runs the Plasma consolidation manager with at most 10 seconds to compute a reconfiguration plan.

We simultaneously run 3 instances of the RUBiS Web application, named `$A1`, `$A2`, and `$A3`. The three tiers of each instance of RUBiS are deployed as 7 VMs (for a total of 21 VMs), as described in Listing 3. Each VM in `$T1` has 512 MB of RAM and runs Apache 2.1.12. Each VM in `$T2` has 1 GB of RAM and runs Tomcat 7.0.2. Finally, each VM in `$T3` has 1 GB of RAM and runs MySQL-cluster 7.2.5, to implement a stateful replicated database. Requests involving dynamic content are distributed by each Apache service to Tomcat services using mod_jk 1.2.28. SQL queries executed in a Tomcat service are distributed to MySQL services using Connector/J 5.1.13. All the VMs are initially deployed in a way that is compatible with their placement constraints.

```
1 //Datacenter description
2 ...
3 $small = {WN[1..4], WN[5..8]};
4
5 //Sample application description
6 $T1 = VM[1..2];
7 $T2 = VM[3..5];
8 $T3 = VM[6..7];
9 spread($T1);
10 spread($T2);
11 spread($T3);
12 latency($T3, $small};
```

Listing 3: Plasma description of the environment.

Recovering from load spikes To study the impact of Plasma on performance, we created a fixed scenario involving three RUBiS applications and a varying number of clients, which we then test without consolidation, with static consolidation, and with dynamic consolidation using Plasma. The number of clients of each application varies over time, as shown in Figure 8. The maximum number of clients that the RUBiS Web application can handle in our setting is 1200. Application `$A2` serves this number of clients around 8 minutes. In the experiment without consolidation, each VM is hosted by a separate

node satisfying its placement constraints.[4] This strawman approach evaluates the maximum performance of the Web applications. In the experiment with static consolidation, we deploy a configuration that is viable with regards to the placement constraints of the VMs and their average resource consumption according to the initial set of clients. In the experiment with dynamic consolidation, Plasma performs reconfiguration when the addition of new clients causes the configuration to become non-viable.

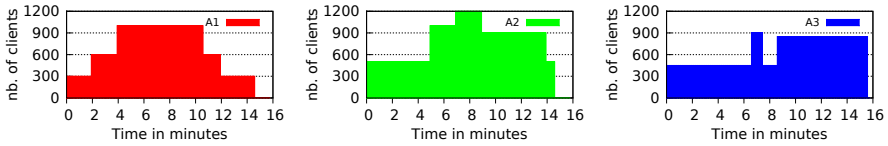


Figure 8: Number of simulated clients during the execution of the experiment for each instance.

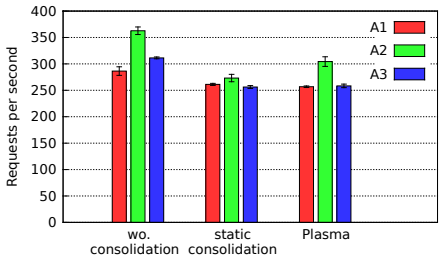


Figure 9: Average throughput of the instances with their associated standard error (maximum 10 req/s.) over 5 repetitions of the experiments.

Figure 9 shows the throughput of each instance in the different experiments. The execution using static consolidation reduces the throughput by 17.7% with respect to no consolidation while it is reduced by only 14.7% using Plasma. Without consolidation, however, 21 nodes are required, while consolidation allows using only 8 nodes, a reduction of 62%.

We also observe that the global throughput with Plasma is 3% higher than using static consolidation. Even if the benchmarks have different loads spikes, there are still periods where nodes cannot provide sufficient uCPU resources to satisfy the demands of their VMs. With static consolidation, this situation leads to a loss of performance. With Plasma however, the non-viable configurations are resolved using migrations. Over the 5 repetitions of the experiment, Plasma performs an average of 12 reconfigurations with an average reconfiguration duration of 29 seconds. The longest reconfiguration required 89 seconds, including the solving time, and 10 VM migrations.

[4]Due to the limited number of nodes available, it was furthermore necessary to run each Web application and its benchmark instances separately. This, however, has no impact on the results, because the intent of the experiment is that the VMs of each application should each run on separate nodes.

Recovering from external events In this experiment, we study the reconfiguration process that restores a cluster to a viable configuration when external events, such as the need for maintenance, occur. We observe the actions that such events entail and the duration of the reconfiguration process.

Time	Event	Reconfiguration
2'10	+ ban({WN8})	3 + 3 migrations in 0'42
4'30	+ ban({WN4})	2 + 7 migrations in 1'02
7'05	- ban({WN4})	no reconfiguration
11'23	+ ban({WN4})	*no solution*
11'43	- ban({WN8}) + ban({WN4})	2 migrations in 0'28

Table 2: External events that occurred during RP-HA. '+' indicates a constraints injection while '-' indicates a removal.

Table 2 summarizes the structure of the experiment. At various times, ban constraints are added (+) or removed (−) to allow maintenance. We simulate node failure, by injecting a ban constraint which results in the same effect. Each time the configuration is observed to be non-viable, the solver is allocated 10 seconds to determine a new configuration. Table 2 shows the number of migrations performed and their duration.

A ban constraint most obviously requires migrating the VMs hosted on the specified nodes to other nodes. Nevertheless, such a constraint can have the side effect of entailing other migrations, when there is not already a sufficient amount of free resources on other nodes for these VMs. This is illustrated at time 2'10, when a constraint ban({WN8}) is injected. The addition of this constraint requires relocating the 3 VMs that WN8 was hosting. But it also requires three additional migrations, to obtain a viable host for all of the VMs. A similar situation occurs at 4'30 when 7 additional migrations were performed in order to be able to migrate the 2 VMs that were running on WN4. Such additional migrations are hard to plan manually.

The events at the end of the experiment illustrate the behavior when the reconfiguration problem is not solvable. At time 11'23, the constraint ban({WN4}) is injected by the administrator. However, the consolidation manager is not able to compute a viable configuration. The datacenter administrator is informed that the maintenance is not currently possible. At time 11'43, the administrator thus removes the ban on WN8 and retries the ban on WN4, which is this time successful, leading to 2 migrations.

4.2 Scalability of Plasma

The difficulty of solving a RP depends on the following parameters: the resource demands (both uCPU and RAM), the specified constraints and the number of VMs and nodes. We evaluate the impact of each of these parameters by generating non-viable configurations of a datacenter hosting HA Web applications, then we analyze the duration of the solving process and the resulting reconfiguration process.

We perform three experiments, first to evaluate the impact of the global uCPU demand, then to evaluate the impact of the constraint types, and finally

to evaluate the impact of the problem size. In the first two experiments, we simulate a datacenter composed of 200 nodes, stacked into 4 racks, that provides two classes of latency, `$small` and `$medium`. This datacenter is specified in Listing 4. In each case, twenty 3-tier applications are hosted (`$A1` to `$A20`), with their maximum resource usage comparable to the standard VMs defined by Amazon EC2 [1]. Each tier of the applications is replicated as described in Listing 5, amounting to 20 replicas per application. A total of 400 VMs are thus hosted by the datacenter. A VM migration is estimated to require 1 second per gigabyte of RAM, while launching a VM is estimated to require 10 seconds. All the VMs are assigned randomly to nodes satisfying their constraints, taking into account their memory requirement but not their uCPU consumption. The uCPU consumption of each VM is then chosen randomly between 0 and its maximum usage, which may induce non-viable configurations. We finally simulate hardware failures, with 1% of the nodes being taken off-line. This can be considered to be a worst-case scenario for the computation of a viable configuration.

```
1 $R1 = WN[1..50]
2 $R2 = WN[51..100];
3 $R3 = WN[101..150];
4 $R4 = WN[151..200];
5 $P1 = $R1 + $R2;
6 $P2 = $R3 + $R4;
7 $small = {$R1, $R2, $R3, $R4};
8 $medium = {$P1, $P2};
```

Listing 4: The simulated datacenter. Nodes in `$R1` and `$R2` have a capacity of 8 uCPU and 32 GB RAM. Nodes in `$R3` and `$R4` have a capacity of 14 uCPU and 48 GB RAM.

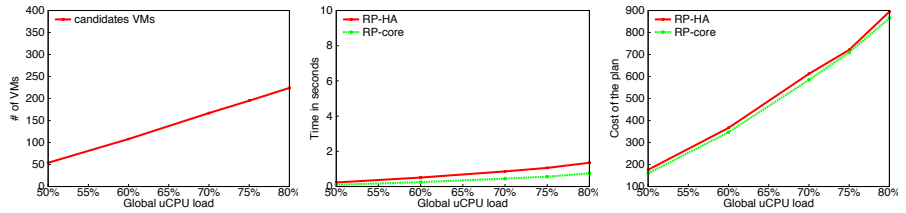
```
1 $T1 = VM[1..5];
2 $T2 = VM[6..15];
3 $T3 = VM[16..20];
4 spread($T1);
5 spread($T2);
6 spread($T3);
7 latency($T3, $medium};
```

Listing 5: A simulated HA Web application. VMs in `$T1` and `$T2` use 7.5 GB RAM and at most 4 uCPU each (*large instances* in the Amazon EC2 terminology). VMs in `$T3` use 17.1 GB RAM and at most 6.5 uCPU each (*high-memory extra-large instances*).

We run the Plasma plan module on a dual processor quad-core Intel Xeon-L5420 at 2.5 GHz and 32 GB RAM, of which we use only one core. The node runs Linux 2.6.26-2-amd64 and Sun's JVM 1.6u21 with 5 GB RAM allocated to the heap at startup. The time to generate the specialized RP is ignored in these experiments as a single RP can be used multiple times when the number of nodes, of VMs, and the set of constraints is unchanged. For each experiment, we allocate a finite amount of time to the plan module to compute the best

reconfiguration plan possible. Three outcomes are possible. Either the plan module finds a solution, or it proves that there is no solution, or it times out without doing either. To evaluate the impact of application specific placement constraints, the plan module is run in two modes: (`RP-Core`) without the application placement constraints, and (`RP-HA`) with the application constraints.

Impact of the global uCPU demand In a first experiment, the plan module is run on configurations with a global application uCPU demand varying between 50% and 80% of the total uCPU capacity of the datacenter. The plan module was allocated 2 minutes to solve the RP. This was repeated 100 times. For all the RPs, Plasma was able to compute at least a solution.



(a) Candidate VMs (b) Solving duration (c) Cost of the plans

Figure 10: Impact of the global uCPU on the solving process.

Figure 10(a) shows the number of VMs selected as candidates by the plan module. We observe that this number increases with the uCPU load. While only 54 running VMs are indeed selected for a load of 50%, 224 VMs (56% of all the running VMs) are selected for a load of 80%. Figure 10(b) shows the average time for the plan module to compute the first solution. We observe that the solving duration slightly increases with the global uCPU load from 0.2 seconds to 1.5 seconds. The reason is that the solver has to compute a host for an increasing number of candidate VMs. The solving duration is slightly longer for `RP-HA` than for `RP-core`. This is due to the additional algorithms associated with the placement constraints that are injected into `RP-HA`. This duration appears, however, to be negligible. Finally, we observe in Figure 10(c) that the cost of the computed reconfiguration plans increases with the global uCPU load. When a larger number of candidate VMs are selected, more VMs could have to be migrated to reach a viable configuration. In addition, when a running VM has to migrate away from an overloaded node, it may not be possible to select a node that already has sufficient free resources. In this situation, additional migrations have to be executed to first free the required resources, increasing the migration cost.

In practice, we find that the first computed solution is almost always the best one. Indeed, out of 500 simulations, there are only 14 cases where the final solution has a lesser cost than the first computed plan. Computing the solution with the lowest possible value of K takes time, and is only worthwhile if the time spent to compute the solution is less than the reduction in K provided by the new solution. For this experiment, 11 of the 14 additional solutions for `RP-HA` provide a reduction in cost that is greater than the added solving time, and 9 of them were computed at most 1 second after the first solution. This

result suggests that 3 seconds is a sufficient timeout value for the plan module solving process with these workloads.

Impact of the constraint types In a second experiment, we observe the impact of the constraints on the solving process. A placement constraint restricts the number of solutions and increases the solving complexity of a RP. Specifically, we consider the impact of adding a `ban`, `fence`, or `latency` constraint. `Ban` and `fence` constraints restrict the pool of nodes available to host VMs. A `latency` constraint with the latency class `$small` provides smaller groups of nodes that must be chosen from .

We set the global uCPU demand to 60% and define three scenarios, each extending `RP-HA` with different constraints. In `RP-HA-sl`, the `latency` constraint (Listing 5, line 7) uses the latency class `$small` instead of `$medium`, as used by `RP-HA`. In `RP-HA-ban`, the nodes `WN0` through `WN11` are made unavailable using a `ban` constraint. In `RP-HA-fence`, two `fence` constraints place applications `$A1` to `$A5` on `$P1`, and applications `$A6` to `$A10` on `$P2`.

Situation	solved	duration	cost
`RP-HA`	100 %	0.3 s	366
`RP-HA-sl`	100 %	0.5 s	384
`RP-HA-ban`	96 %	0.7 s	1160
`RP-HA-fence`	51 %	0.3 s	619

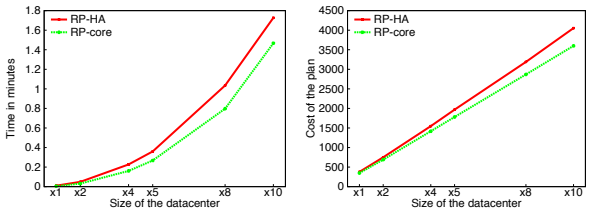
Table 3: Impact of the constraints on the solving process.

`RP-HA-sl`, `RP-HA-ban`, and `RP-HA-fence`, all restrict the number of nodes that can host particular VMs in the final configuration. `RP-HA-ban` and `RP-HA-fence` have a higher failure rate than `RP-HA`. This is partially explained by the heuristic that selects the candidate VMs. The selection is too restrictive in some cases for the solver to be able to compute a viable configuration. Indeed, changing the selection heuristic so that it selects all the running VMs as candidates improves the success rate to 100% for `RP-ban` and 65% for `RP-fence`. An approach to improve the solving capability of the plan module for such strongly restricted RPs could be to use an incremental solving process that considers more candidate VMs when the solver fails to compute a solution. The other reason that explains the high failure rate of `RP-HA-fence` is that in some cases, there are simply too few available nodes to perform a successful reconfiguration. Thus fencing, even with half of the available nodes within each fence, is a bad idea in practice on such datacenter configurations. These results demonstrate the usefulness of simulations to allow the datacenter administrators to plan the usage of the system.

Solutions found for `RP-HA-ban`, and `RP-HA-sl` have an higher cost than those found for `RP-HA`. Indeed, in the case of `RP-HA-ban`, VMs on the excluded nodes have to be migrated, while for `RP-HA-sl`, the solver has to perform extra migrations to gather all the VMs of each stateful tier to one group of nodes in the latency class `$small`.

Impact of the problem size Finally, we study the impact of the size of the problem on the duration of the solving process. We have defined 6 sets of

configurations that differ in the number of VMs and nodes. The set `x1` defines the standard simulated datacenter, with 400 VMs and 200 nodes. The sets `x2` to `x10` are 2 to 10 times the size of the set `x1`. The set `x10` is then composed of configurations with 2000 nodes and 4000 VMs. The global uCPU demand is set to 60%. We give the solver 10 minutes in order to allow it to compute a solution in each of the modes `RP-Core` and `RP-HA`. For both `RP-Core` and `RP-HA` at least one solution was computed for all the problems.



(a) solving duration (b) cost of the solutions

Figure 11: Impact of infrastructure size on the solving process.

Figures 11(a) and 11(b) show the impact of the size of the problem on the time required to obtain the first solution, and on the cost of these solutions, respectively. We observe that the solving durations of the RPs and the costs of the resulting plans increase significantly with the number of VMs and nodes. The increase is however faster for `RP-HA` than for `RP-Core`. For the largest sizes, solving a `RP-HA` takes 15 additional seconds with a costs higher by 12%. This increase is due to the large amount of placement constraints to consider in `RP-HA` which reduce the ability to leave VMs on their current nodes.

We also observe that the increase of the solving duration appears to be exponential, while the number of candidates VMs increases linearly. This situation is unsurprising for an NP-Hard problem. However, the solving duration of the problems in `RP-HA` for the set `x10` is still reasonable considering the number of variables and constraints that must be considered. Indeed, it takes to the solver ony an average of 100 seconds to solve an `RP-HA` problem, which involves 1) selecting a host among the 1980 online nodes for an average of 1117 candidate VMs, while taking into account the nodes' resource capacity, 600 `spread` constraints, and 200 `$medium latency` constraints, and 2) planning the execution of the 475 actions.

5 Extending Plasma with new constraints

The constraints `spread` and `latency` provide an application administrator with the foundations for a viable deployment of an HA application. Similarly, the constraints `ban` and `fence` provide a datacenter administrator with tools for common system maintenance tasks. Nevertheless, in a datacenter with a focus on different concerns, it may be useful to provide additional constraints. For example, placement constraints may be used to prevent disk I/O bottlenecks when VMs disk images are stored on the hosting nodes rather than on remote file servers. A datacenter administrator may also want to constrain the placement of the VMs to balance the network bandwidth usage.

Contrary to a rule based engine or an *adhoc* heuristic, the plan module of Plasma can be easily extended as it provides a deterministic composition of the placement constraints. Each placement constraint of Plasma is already considered as a plugin, loaded on demand, that contains the signature of the constraint for the configuration language and its implementation using the API of the constraint solver. A developer can thus model a new placement constraint in the same way, and link this model to the variables of the core RP just as the `latency` constraint.

6 Related Work

Dynamic consolidation Nathuji et al. [16] present power efficient mechanisms to control and coordinate the effects of various power management policies. This includes the packing of VMs through live migration. Bobroff et al. [6] base their reconfiguration engine on a forecast service that predicts, for the next forecast interval, the resource demands of VMs, according to their history. Then the reconfiguration algorithm selects a node than can host the VMs during this time interval. To ensure efficiency, the forecast window takes into account the duration of the reconfiguration process. Verma et al. [23] additionally consider a power model to select the hosting nodes according to their power consumption. Wood et al. [26] exploit the page sharing between the VMs to improve the packing. Finally, Yazir et al. [28] present a consolidation manager that takes into account a variable amount of weighted placement criteria related to the resource consumption of a VM: memory, CPU usage, latency, etc. Computing the placement of the VMs is decomposed into independent tasks, performed by multiple autonomous agents to improve scalability. All of these works provide heuristics that cannot be specialized with additional placement constraints that are required by application administrators and datacenter administrator.

DRS [25] is a resource manager from VMWare that can be used to perform dynamic consolidation. It provides to the datacenter administrator a feature similar to the `ban` constraint and an *affinity rule* similar to `spread`. `Latency` and `fence`, however, are unavailable. Application administrators can not declare themselves their placement requirements. In addition, actions performed by the datacenter administrator can supersede the inserted rules. This feature is error-prone as the administrator must manually ensure that the actions are compatible with all of the stated rules. Finally, to the best of our knowledge, rules are implemented as standalone heuristics that are considered individually. This limits the ability of DRS to compute reconfiguration plans that migrate additional VMs to solve complex configuration problems. In contrast, we provide an approach that simultaneously considers all the constraints to compute a globally optimized solution. Hermenier et al. [12] provide an approach based on constraint programming to place VMs. Nevertheless, the reconfiguration algorithm is partially based on a heuristic that cannot handle additional placement constraints such as `spread`. Finally, while Plasma selects only a small fraction of the running VMs as candidates to repair a non-viable configuration, Entropy considers all the running VMs, which limits its scalability to a few hundred of VMs and nodes.

Management of virtualized multi-tier applications Urgaonkar et al. [21] propose a provisioning module to estimate the optimal number of replicas to run per tier to satisfy a workload. Each node hosts replicas of multiple tiers, but only one of these replicas is active at a given time. Depending on the intensity of each tier's workload, a controller adjusts the ratio of activated replicas of the various tiers. Pradeep et al. [17] consider a datacenter that simultaneously hosts several multi-tier applications. VMs are statically placed by the datacenter administrator on the nodes. Then, a resource controller dynamically adjusts the distribution of resources to the individual tiers to meet the Service Level Agreements (SLA) of the application.

Jung et al. [13] compute the number of replicas and the configuration offline. In addition, they use the SLAs to generate rules that are used online to adapt the scheduling policies of the VMM depending on the current resource demands of the VMs. Later, they extended their approach to generate relocation rules for the VMs to solve resource contention that take into account the cost of the migrations [14].

These works focus on resource control policies so as to adapt the application structure to the workload. However, they do not consider HA placement constraints.

7 Conclusion and Future Work

Consolidation of VMs allows multiple applications to share nodes within a datacenter. However, modern applications have scalability and high availability requirements, and reconciling these requirements while allowing node sharing is challenging. We have proposed Plasma, a configurable consolidation manager allowing datacenter and application administrators to describe placement constraints. Configuration scripts are interpreted on the fly to customize a core reconfiguration algorithm. The resulting specialized reconfiguration algorithm is then able to rearrange the placement of the VMs when the VMs do not have access to sufficient resources.

Experiments on simulated data show that introducing placement constraints has a limited impact on the execution time of the reconfiguration algorithm. Our implementation can compute reconfigurations involving up to 4000 VMs, 2000 nodes and 800 constraints less than 2 minutes. Real experiments on a cluster with 8 working nodes running 3 instances of the RUBiS benchmarks with a total of 21 VMs show that continuous consolidation allows this cluster to reach 85% of the load of a cluster with 21 working nodes.

In future work, we propose to improve the simulation mode of the plan module so that a datacenter administrator may use it as a planning tool. We also plan to integrate additional types of constraints, *e.g.* on power consumption, so as to be able to optimize the datacenter's use of energy when the application demand is low. We also plan to consider placement constraints that can be violated with a penalty expressed using high-level metrics. Finally, we plan to improve the scalability of the plan module by detecting independent subproblems that can be solved in parallel using multiple cores.

Acknowledgment

Experiments on simulated data presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies[5].

References

[1] Amazon EC2 Instance Types. `http://aws.amazon.com/ec2/instance-types/`.

[2] Anderson, E., Hall, J., Hartline, J., Hobbes, M., Karlin, A., Saia, J., Swaminathan, R., and Wilkes, J. Algorithms for data migration. *Algorithmica 57* (2010), 349–380. 10.1007/s00453-008-9214-y.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles* (Oct. 2003), ACM Press, pp. 164–177.

[4] Beldiceanu, N., and Carlsson, M. A new multi-resource cumulatives constraint with negative heights. In *CP '02: 8th International Conference on Principles and Practice of Constraint Programming* (2002), Springer-Verlag, pp. 63–79.

[5] Beldiceanu, N., Carlsson, M., and Rampon, J.-X. Global constraint catalog. Tech. Rep. T2005-08, Swedish Institute of Computer Science, 2005.

[6] Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing SLA violations. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on* (May 2007), 119–128.

[7] Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoel, W. Performance comparison of middleware architectures for generating dynamic web content. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware* (New York, NY, USA, 2003), Springer-Verlag New York, Inc., pp. 242–261.

[8] Choco: an open source Java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[9] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)* (May 2005), pp. 273–286.

[10] Hall, J., Hartline, J., Karlin, A. R., Saia, J., and Wilkes, J. On algorithms for efficient data migration. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2001), SODA '01, Society for Industrial and Applied Mathematics, pp. 620–629.

[11] Hermenier, F., Lèbre, A., and Menaud, J.-M. Cluster-wide context switch of virtualized jobs. In *VTDC10 - The 4th International Workshop on Virtualization Technologies in Distributed Computing* (June 2010).

[12] Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., and Lawall, J. Entropy: a consolidation manager for clusters. In *VEE '09: 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2009), ACM, pp. 41–50.

[13] Jung, G., Joshi, K. R., Hiltunen, M. A., Schlichting, R. D., and Pu, C. Generating adaptation policies for multi-tier applications in consolidated server environments. In *ICAC '08: 2008 International Conference on Autonomic Computing* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 23–32.

[14] Jung, G., Joshi, K. R., Hiltunen, M. A., Schlichting, R. D., and Pu, C. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Middleware '09: 10th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2009), Springer-Verlag, pp. 1–20.

[5] https://www.grid5000.fr

[15] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7 (2004), 817 – 840.

[16] Nathuji, R., and Schwan, K. VirtualPower: Coordinated power management in virtualizaed entreprise systems. In *21st Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

[17] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 289–302.

[18] Rossi, F., van Beek, P., and Walsh, T. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[19] Ruth, P., Rhee, J., Xu, D., Kennell, R., and Goasguen, S. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on* (2006), pp. 5–14.

[20] Shaw, P. A constraint for bin packing. In *Principles and Practice of Constraint Programming (CP'04)* (2004), vol. 3258 of *Lecture Notes in Computer Science*, Springer, pp. 648–662.

[21] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst. 3*, 1 (2008), 1–39.

[22] van Hoeve, W. J. The alldifferent constraint: A survey. *CoRR cs.PL/0105015* (2001).

[23] Verma, A., Ahuja, P., and Neogi, A. pMapper: power and migration cost aware application placement in virtualized systems. In *Middleware '08: 9th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2008), Springer-Verlag New York, Inc., pp. 243–264.

[24] Virtualization management index. Tech. rep., VKernel, Dec. 2010.

[25] VMWare Infrastructure: Resource Management with VMWare DRS. Tech. rep., 2006.

[26] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *VEE '09: 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2009), ACM, pp. 31–40.

[27] Yang, L., Foster, I., and Schopf, J. M. Homeostatic and tendency-based CPU load predictions. In *IPDPS '03: 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 42.2.

[28] Yazir, Y., Matthews, C., Farahbod, R., Neville, S., Guitouni, A., Ganti, S., and Coady, Y. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. pp. 91 –98.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399