RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Consolidation dynamique d'applications Web haute disponibilité

Fabien Hermenier[1], Julia Lawall[2], Jean-Marc Menaud[1], Gilles Muller[3]

[1]ASCOLA Mines de Nantes, INRIA, LINA. prenom.nom@mines-nantes.fr
[2]DIKU, Université de Copenhague. julia@diku.dk
[3]INRIA/LIP6-Régal. gilles.muller@lip6.fr

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 réalisée par Plasma atteint 85% des performances d'une grappe de 21 nœuds.

Mots-clés : Virtualisation, centre de données, placement, haute disponibilité, consolidation dynamique

1. Introduction

Les applications Web modernes telles que Facebook, Twitter, ou eBay, sont structurées comme des services n-tiers comprenant, entre autre, un serveur HTTP, un serveur d'applications et une base de données. Pour être hautement disponible, ces applications doivent s'adapter à la charge et tolérer les pannes. Ces objectifs sont atteints en répliquant leurs tiers. Dans la pratique, cette réplication n'est cependant pas suffisante. La tolérance aux pannes est en effet effective uniquement lorsque les réplicas des tiers sont placés sur des nœuds distincts. De plus, les réplicas d'un service peuvent communiquer entre eux pour se synchroniser. Ils doivent alors être placés sur des nœuds interconnectés par un réseau ayant une latence et un débit adaptés au protocole de synchronisation. Ces contraintes de placement peuvent être exprimées par un contrat de service, décrit par l'administrateur de l'application et fourni avec l'application lorsqu'elle est soumise à l'administrateur système du centre de données.
Pour un contrat de service donné, l'administrateur système du centre de données se doit d'assurer que les contraintes de placement seront satisfaites durant l'exécution de l'application, malgré la variabilité de la disponibilité des ressources, due à des pics de charge, des opérations de maintenance ou des pannes. Une solution serait de placer chaque réplica sur son propre nœud, choisi selon les contraintes de placement. La charge moyenne d'un réplica est cependant bien inférieure à la capacité d'un nœud et une telle approche diminuerait la capacité d'hébergement de l'infrastructure et son efficacité énergétique.
En consolidant les réplicas de différents tiers sur un même nœud, la capacité d'hébergement du centre de données augmente. Exécuter les réplicas dans des machines virtuelles (VM) assure alors leur isolation.

La migration à chaud [5] permet finalement de les ré-agencer sur d'autres nœuds lorsque leur charge augmente et sature leur nœud d'hébergement [13]. Un gestionnaire de consolidation détermine alors quand ré-agencer les VM, quelles VM doivent être migrées et quels sont les nœuds qui doivent les héberger. Définir un tel système est complexe car il doit satisfaire à la fois les objectifs de l'administrateur système et des administrateurs des applications. La plupart des approches actuelles [4, 7, 10, 13, 14, 17] optimisent le placement des VM uniquement d'après leur consommation en ressources. VMWare DRS [15] quant à lui, autorise l'administrateur système à définir des règles d'affinité entre des VM, permettant uniquement de les placer sur un même nœud ou des nœuds distincts.

Nous proposons dans ce papier un gestionnaire de consolidation, Plasma, qui propose un algorithme de reconfiguration spécialisable pour considérer, en plus des contraintes liées à la consommation en ressources des VM, les contraintes de placement des applications hautement disponible. La spécialisation est réalisée par l'intermédiaire de scripts de configuration qui permettent à l'administrateur système de décrire l'infrastructure du centre de données et aux administrateurs des applications de décrire les contraintes de placement de leur VM. Notre approche fournit une consolidation dynamique 1) garantissant aux administrateurs des applications que leur contraintes de placement seront satisfaites, 2) relevant l'administrateur système des problèmes liés à la compatibilité des opérations de maintenance avec les contrats de service. Nos principaux résultats sont :

– Un algorithme de reconfiguration extensible qui considère uniquement une estimation des VM mal placées pour calculer une reconfiguration. Cette approche permet à Plasma d'être effectif pour des centres de données composés de milliers de nœuds et de VM.
– Un déploiement sur une grappe de 8 nœuds. L'exécution de 3 instances de l'application RUBiS sur 21 VM montre la réactivité de Plasma pour ré-agencer les VM lorsque des nœuds sont surchargés, et permet d'obtenir 85% des performances d'une grappe de 21 nœuds.
– Des expérimentations sur des données simulées montrent que notre implantation peut superviser 2000 nœuds hébergeant 4000 VM avec 800 contraintes de placement et résoudre les problèmes de reconfiguration associés en moins de 2 minutes.

La suite de ce papier est organisée comme suit. La section 2 décrit l'architecture globale de Plasma. La section 3 décrit l'implantation de notre algorithme de reconfiguration. La section 4 évalue notre prototype. Finalement, la section 5 décrit les travaux apparentés et la section 6 présente nos conclusions et travaux futurs.

2. Architecture de Plasma

L'objectif de Plasma est de choisir un placement pour les réplicas des tiers des applications satisfaisant un ensemble de contraintes décrites par des scripts de configuration. Dans cette section, nous présentons d'abord les scripts de configuration, puis l'architecture du gestionnaire de consolidation.

2.1. Scripts de configuration

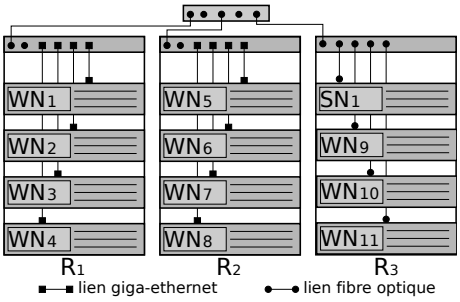
Les scripts de configuration permettent à l'administrateur système et aux administrateurs des applications de décrire leur vue du centre de données et de leurs applications, respectivement. Notre approche (1) permet à l'administrateur système de gérer ses nœuds sans connaissance des contraintes de placement des applications, (2) permet à l'administrateur d'une application d'exprimer ses contraintes de placement sans une connaissance détaillée de l'infrastructure.

Quatre contraintes permettent ensuite de restreindre le placement des VM : `ban` et `fence` sont à disposition de l'administrateur système pour exprimer des contraintes liées à des tâches d'administration tandis que `spread` et `latency` permettent à l'administrateur d'une application d'exprimer des contraintes de placement entre ses VM.

Description d'un centre de données

L'administrateur système décrit les nœuds disponibles, leur rôle et leur connectivité. Un centre de données virtualisé est constitué d'un ensemble de nœuds de calcul et de nœuds de service. Les nœuds de calcul hébergent les VM des applications grâce à un hyperviseur (VMM) tel que Xen [2]. Les nœuds de service exécutent les applications de gestion de l'infrastructure (supervision, serveurs de fichiers, etc.). Tous les nœuds sont empilés dans des armoires et interconnectés par un réseau hiérarchique (Figure 1a),

fournissant une latence non-uniforme à l'intérieur de l'infrastructure.



(a) Infrastructure

```
1  $R1 = {WN1, WN2, WN3, WN4};
2  //Definition par une suite d'elements
3  $R2 = WN[5..8];
4  //Definition par une union
5  $R3 = WN[9..11] + {SN1};
6  $small = {$R3};
7  $medium = {$R1, $R2, $R3};
8
9  ban($ALL_VMS, {SN1});
10 ban($ALL_VMS, {WN5});
11 fence($A1, $R2 + $R3);
```

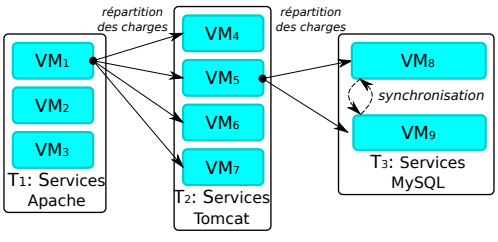
(b) Script de configuration Plasma

FIGURE 1: Exemple d'infrastructure d'un centre de données regroupant des nœuds de calcul (WN) et des nœuds de service (SN), et description des contraintes de placement associées.

La description du centre de données fournit par son administrateur est illustrée entre les lignes 1 et 7 du script de configuration du Listing 1b. Les lignes 1 à 5 définissent les variables `$R1` à `$R3` comme la liste des nœuds des armoires R_1 à R_3, respectivement. Les lignes 6 et 7 définissent les différentes classes de latence disponible. Une telle classe est définie par un ensemble disjoint de groupes de nœuds. Ici, la classe `$small` est composée des nœuds connectés par de la fibre optique tandis que la classe `$medium` est composée des groupes de nœuds ayant un seul commutateur entre eux. La totalité des VM est sélectionnable par la variable `$ALL_VMS`. Une interdiction d'hébergement de VM sur des nœuds s'exprime par la contrainte `ban`. Ici cette contrainte permet d'assurer que les nœuds de service n'hébergeront pas d'applications (ligne 9) et permet d'isoler le nœud `WN5` durant une maintenance (ligne 10). Finalement, il est également possible de restreindre l'hébergement de VM à un ensemble de nœuds avec la contrainte `fence`. Pour chaque application hébergée sur le centre de données, Plasma génère une variable spécifiant la totalité des VM de l'application. À la ligne 11, la variable `$A1` identifie l'ensemble des VM de l'application `A1`, et la contrainte `fence` associée spécifie que les VM de `$A1` doivent s'exécuter sur les nœuds des armoires R_1 et R_2.

Description d'une application

L'administrateur d'une application spécifie les VM de son application et les contraintes de placement qui devront être satisfaites. La Figure 2a illustre une application Web 3-tiers. Les services Apache et Tomcat sont sans états : chaque requête est indépendante et aucune synchronisation n'est nécessaire entre les réplicas. À l'opposé, le service MySQL est composé de bases de données répliquées et les transactions modifiant les données d'un réplica doivent être propagées aux autres réplicas pour maintenir un état global consistant. Pour être tolérant aux pannes, les VM d'un même tiers doivent être hébergées sur des nœuds distincts. Pour des questions de performance, les VM d'un service à état global doivent être hébergées sur des nœuds ayant une latence adaptée au protocole de synchronisation des états locaux.



(a) Architecture de l'application

```
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, $medium);
8 //Variable generee par Plasma
9 //$A1 = VM[1..9];
```

(b) Script de configuration Plasma

FIGURE 2: Architecture de l'application Web 3-tiers `$A1` et description des contraintes de placement associées.

Le Listing 2b présente le script de configuration de l'application de la Figure 2a. L'administrateur décrit, de la ligne 1 à 3, la structure de l'application en définissant une variable par tiers. Les lignes 4 à 6 assurent

la tolérance aux pannes de chaque tiers. Finalement, la ligne 7 assure une synchronisation efficace du tiers T_3 en demandant un placement de ces VM sur un des groupes de nœuds de la classe `$medium`.

2.2. Architecture du gestionnaire de consolidation

L'objectif du gestionnaire de consolidation de Plasma est de maintenir le centre de données dans une *configuration*, i.e. une affectation des VM aux nœuds, qui est *viable*, c'est à dire que toutes les VM ont accès à suffisamment de ressources, et sont placées de manière à satisfaire toutes les contraintes de placement spécifiées. Plasma s'exécute sur un nœud de service et initie une réaffectation des VM lorsqu'il détecte que la configuration courante n'est plus viable. Le gestionnaire de consolidation s'articule autour d'une boucle de contrôle infinie composée de quatre modules (Figure 3).

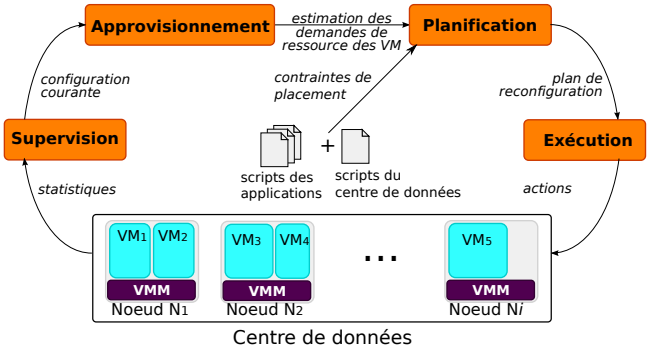


FIGURE 3: Boucle de contrôle de Plasma.

Module de supervision

Ce module utilise le système de supervision Ganglia [1] pour connaitre l'état courant de chaque nœud et VM, le placement des VM et leur consommation CPU et mémoire, ainsi que les capacités CPU et mémoire de chaque nœud. La ressource CPU est exprimée dans une unité indépendante du matériel, appelée uCPU, qui reprend le principe des instances d'Amazon EC2 [1] [2]. Les statistiques relatives aux éléments supervisés sont récupérées toutes les 20 secondes. Lorsque le Ganglia ne reçoit pas de mise à jour des statistiques d'un élément, celui-ci est considéré défaillant.

Module d'approvisionnement

Ce module estime les besoins uCPU des VM en se référant aux statistiques du module de supervision. La consommation uCPU d'un réplica peut varier au cours du temps. Le module prédit alors les nouveaux besoins d'une VM en observant les changements récents de sa consommation [16].

Module de planification

Ce module utilise les scripts de configuration fournis par l'administrateur système et les administrateurs des applications, les nouveaux besoins uCPU des VM et leur état courant pour statuer sur la viabilité de la configuration courante. Si toutes les VMs sont en cours d'exécution, disposent de suffisamment de ressources et sont placées selon les contraintes de placement, alors la configuration courante est viable et la boucle de contrôle est relancée. Sinon, le module calcule une nouvelle configuration viable pour les VM ainsi qu'un *plan de reconfiguration*, une série d'actions de migration et de lancement de VM, qui va placer les VM sur les nœuds indiqués par la nouvelle configuration. L'exécution des actions est planifiée pour assurer la faisabilité de la reconfiguration d'après une estimation de leur durée.

Module d'exécution

Ce module applique un plan de reconfiguration en exécutant les actions associées. Comme le lancement de certaines actions dépend de la terminaison d'autres actions, le plan de reconfiguration est adapté pour éviter l'échec de la reconfiguration lorsque la durée réelle d'une action dépasse son estimation.

1. http ://ganglia.info/
2. L'estimation de la capacité uCPU d'un nœud n'est pas adressée dans ce papier.

3. Implantation du module de planification

Le module de planification repose sur un *algorithme de reconfiguration premier* qui considère uniquement les besoins mémoire et uCPU des VM. Son implantation utilise la Programmation Par Contraintes [12] (PPC). Cette approche permet de spécialiser facilement l'algorithme de reconfiguration premier en ajoutant les contraintes de placement des scripts de configuration.
Dans cette section, nous présentons d'abord brièvement la PPC. Nous décrivons ensuite l'implantation de l'algorithme de reconfiguration premier puis l'implantation des contraintes de placement. Finalement nous présentons notre approche pour réduire le temps de calcul des plans de reconfiguration.

3.1. La programmation par contraintes

La Programmation Par Contraintes (PPC) est une approche complète pour modéliser et résoudre des problèmes combinatoires. Cette approche calcule une solution optimale, si elle existe, par un parcours pseudo-exhaustif d'un arbre de recherche. La PPC modélise un problème en définissant des contraintes (des relations logiques) devant être satisfaites par la solution. Un problème de satisfaction de contraintes (CSP) est défini par un ensemble de variables, chacune prenant sa valeur dans un domaine défini, et un ensemble de contraintes indépendantes qui restreignent les valeurs que peuvent prendre les variables.
Un solveur calcule une solution d'un CSP en affectant une valeur à chaque variable satisfaisant simultanément toutes les contraintes. L'algorithme de résolution d'un CSP est générique et indépendant des contraintes qui le compose. Dans notre cas, la PPC fournit une spécialisation deterministe de l'algorithme de reconfiguration premier et un processus de résolution déterministe même en présence de contraintes manipulant les même variables. L'algorithme de reconfiguration est modélisé par des contraintes standard [3], disponible dans différents solveurs. Dans la pratique, Plasma est écrit en Java et utilise le solveur de contraintes Choco [3].

3.2. Algorithme de reconfiguration premier

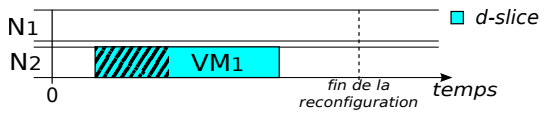
Calculer une configuration viable nécessite de choisir un nœud pour chaque VM satisfaisant ses besoins en ressource, et d'ordonnancer l'exécution des actions qui convertiront la configuration courante en celle calculée. Nous nommons ce CSP *problème de reconfiguration* (RP). Lorsqu'une reconfiguration est nécessaire, le module de planification utilise la configuration courante et l'estimation des besoins uCPU des VM du module d'approvisionnement pour générer un RP grâce à l'API de Choco. Le module insère ensuite dans ce RP *premier*, les contraintes de placement des scripts de configuration. Le RP spécialisé est alors résolu pour calculer un plan de reconfiguration.
Un RP est défini par un ensemble de nœuds et de VM. Chaque nœud dispose de sa capacité uCPU et mémoire. Durant une reconfiguration, une VM en cours d'exécution peut rester sur le même nœud ou être placée sur un autre nœud. Il se produira alors une migration. Une VM en attente d'exécution peut être placée sur un nœud. Il se produira alors une action de lancement. Une *slice* est une période durant une reconfiguration durant laquelle une VM s'exécute sur un nœud et utilise une portion constante de ses ressources. Les slices sont utilisées dans un RP pour modéliser la consommation en ressources des VM durant une reconfiguration. Nous distinguons les *c-slices* et les *d-slices*.
Une *c-slice* est une période durant laquelle une VM s'exécute déjà sur un nœud au début de la reconfiguration. D'après la configuration courante, une c-slice est affectée au nœud hébergeant la VM et consomme une quantité de ressource égale à sa consommation courante. Une c-slice commence nécessairement au début de la reconfiguration mais sa date de fin n'est pas arrêtée.
Une *d-slice* est une période durant laquelle une VM s'exécutera sur un nœud à la fin de la reconfiguration. Une d-slice n'est pas pré-affectée à un nœud et consomme une quantité de ressource égale à la demande calculée par le module d'approvisionnement. Une d-slice se termine nécessairement à la fin d'une reconfiguration mais sa date de début n'est pas arrêtée.

Modélisation des actions

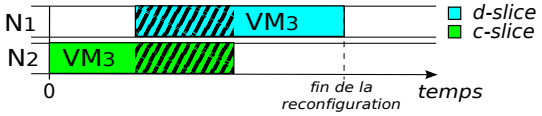
L'affectation d'une VM à un nœud peut produire une action qui s'exécutera durant la reconfiguration. La durée de l'action peut être estimée et modélisée depuis des expérimentations [6]. L'administrateur système fournit alors avec deux fonctions de coûts, une estimation de la durée des actions en considérant

3. http ://choco.emn.fr

la quantité mémoire et uCPU consommée par la VM à manipuler.
L'affectation d'une VM en attente sur un nœud est modélisée avec une d-slice et produira une action de lancement. La Figure 4a illustre, avec un diagramme de Gantt, l'action de lancement d'une VM. Au démarrage de l'action, le VMM alloue la mémoire pour la VM et la démarre. Comme une d-slice s'étend jusqu'à la fin de la reconfiguration, sa durée peut dépasser la durée estimée de l'action.
L'activité d'une VM en cours de fonctionnement est modélisée avec une c-slice. Une d-slice permet ensuite de modéliser sa position à la fin de la reconfiguration. Si ces deux slices ne sont pas placées sur le même nœud, il se produira une action de migration. Le délai entre le début de la d-slice et la fin de la c-slice correspond à la durée estimée de l'action. La Figure 4b illustre la migration de la VM `VM3` du nœud `N2` au nœud `N1`. Lorsque la migration commence, le VMM sur `N1` alloue la mémoire pour `VM3` et commence à copier les pages mémoire tandis que `VM3` continue de s'exécuter sur `N2`. Lorsque la migration est terminée, le VMM sur `N2` libère les ressources et `VM3` est activée sur `N1`.



(a) Affectation de la VM en attente `VM1` produisant une action de lancement. L'action démarre au début de la d-slice.



(b) Ré-affectation de la VM en cours d'exécution `VM3`, produisant une action de migration. L'action démarre au début de la d-slice et se termine à la fin de la c-slice.

FIGURE 4: Modélisation des actions manipulant une VM. La zone hachurée indique la durée de l'action.

Calculer la solution d'un RP consiste à affecter un nœud à chaque d-slice et à calculer son début. Ces valeurs permettront de former le plan de reconfiguration en indiquant la liste des actions à exécuter et un moment pour démarrer chacune d'entre elles.

Calcul d'un plan de reconfiguration vers une configuration viable

Pour satisfaire les besoins en ressource des VM, à aucun moment le cumule des ressources consommées par les slices affectées à un nœud ne doit dépasser sa capacité. Cette restriction est assurée pour la fin de la reconfiguration par 2 contraintes *bin-packing* (une par type de ressource) se focalisant sur les d-slices.
Un nœud peut être la source d'actions sortante (migrations), et la cible d'actions entrante (migrations ou lancements). Pour assurer la faisabilité de la reconfiguration, chaque action entrante doit s'exécuter lorsqu'il y a suffisamment de ressources uCPU et mémoire disponible sur son nœud cible. Cela implique qu'il peut être nécessaire d'exécuter des actions sortantes avant des actions entrantes de manière à libérer des ressources sur un nœud. Pour planifier l'exécution de ces actions, nous avons développé une contrainte inspirée de la contrainte *cumulative*. Dans la pratique, notre contrainte s'assure que le moment de démarrage de chaque action entrante est suffisamment retardé pour assurer la disponibilité des ressources uCPU et mémoire requises, pouvant être libérer à la fin d'actions sortante.
Exécuter une reconfiguration peut prendre un temps non négligeable, durant lequel les performances des applications hébergées sont réduites si les nœuds impliqués n'ont pas suffisamment de ressources libre pour exécuter les actions sans pénaliser les VM. De plus, la durée de migration d'une VM augmente avec sa consommation mémoire. Finalement, retarder une migration réduisant la charge d'un nœud saturé maintient une configuration non-viable qui dégrade les performances des applications. Le coût d'un plan de reconfiguration est défini comme la somme des moments de fin des actions. Ce coût considère ainsi la quantité d'actions à exécuter, leur temps d'exécution et le délai avant leur exécution. En minimisant ce coût, le solveur calculera alors le plan de reconfiguration considéré le plus efficace.

3.3. Implantation des contraintes de placement de Plasma

Les contraintes de placement sont modélisées d'après les variables du RP et des contraintes standard fournies par Choco.
`Spread` empêche la création de points de défaillance en assurant que les VM spécifiées dans la contrainte ne seront jamais hébergées sur un même nœud, même temporairement durant le processus de reconfiguration. Son implantation repose d'abord sur une contrainte *allDifferent* qui force les d-slices à être affectées sur des nœuds distincts. Pour éviter un chevauchement temporaire des slices durant la reconfiguration, des contraintes *implies* retardent ensuite l'arrivée d'une d-slice sur un nœud tant que les

c-slices des autres VM impliquées dans la contrainte ne sont pas terminées.
`Latency` force un ensemble de VM à être hébergé sur un groupe de nœuds unique appartenant à la classe spécifiée en paramètre. Son implantation ajoute au RP premier une nouvelle variable décrivant le groupe de nœuds qui hébergera les VM. Cette variable est alors reliée aux d-slices des VM par une contrainte *element*, indiquant que si la d-slice d'une VM du groupe est affectée à un nœud x, alors toutes les d-slices des autres VM seront placées sur des nœuds du groupe de x.
`Fence` force un ensemble de VM à être hébergé sur le groupe de nœuds spécifié. Cette contrainte est implantée par une restriction de domaine : tous les nœuds qui ne sont pas spécifiés dans la contrainte sont retirés du domaine des variables d'affectation des d-slices.
`Ban` empêche un ensemble de VM d'être hébergé sur un ensemble de nœuds. Elle est l'opposée de la contrainte `fence`. Son implantation repose également sur une restriction de domaine : tous les nœuds spécifiés dans la contrainte sont retirés du domaine des variables d'affectation des d-slices.

3.4. Optimisation du processus de résolution

Calculer une solution d'un RP peut nécessiter un temps significatif. Sélectionner un nœud pour chaque VM est en effet un problème NP-difficile. Notre approche pour réduire le temps de calcul d'un RP consiste en l'utilisation d' une heuristique qui réduit le nombre de VM en cours d'exécution à considérer. Une fois que le RP est spécialisé par les contraintes de placement, chacune d'entre elles vérifie la viabilité de la configuration courante et calcule en cas d'échec, un ensemble de VM mal-placées à reconsidérer pour pouvoir calculer une solution. Nous désignons ces VM comme *candidates*. Les d-slices des VM non sélectionnées sont alors directement affectées à leur nœud courant avant de commencer la résolution du RP. Il résulte alors un RP simplifié, sous-problème du RP d'origine, avec un nombre de d-slices à placer et d'actions à ordonnancer réduit. La visibilité d'une heuristique de sélection des VM candidates est limitée à sa contrainte, l'ensemble des VM candidates peut alors être trop restrictif car il n'y a pas de preuves qu'il soit suffisant pour assurer la solvabilité du RP simplifié. Les heuristiques développées retournent alors actuellement un ensemble de VM candidates plus grand que l'ensemble supposé minimal.

4. Évaluation de Plasma

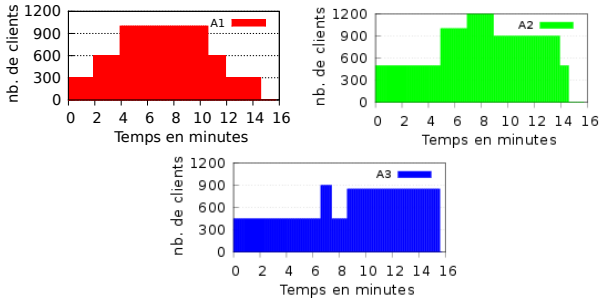
L'objectif de Plasma est d'assurer la consolidation de VM tout en respectant des contraintes de placement. Le RP étant un problème NP-difficile, son temps de calcul peut limiter l'intérêt de notre approche. Dans cette section, nous démontrons d'abord la viabilité de Plasma par un déploiement sur une infrastructure réelle. Nous présentons ensuite, par simulations, la capacité de passage à l'échelle de Plasma.

4.1. Evaluation de Plasma pour l'application RUBiS

RUBiS est une suite logicielle composée d'une application Web d'enchère 3-tiers (type *eBay.com*) et d'une application de test de charge. Une instance de cette application simule un nombre de clients prédéfini, chargés d'émettre des requêtes à l'application Web. Le résultat du test le nombre moyen de requêtes par seconde traitées avec succès par l'application Web.
Notre infrastructure de test se compose de huit nœuds de travail et de quatre de service, tous connectés par un réseau gigabit. Chaque nœud possède 4 GB de mémoire, un processeur Intel Core 2 Duo à 2.1 GHz et exécute l'hyperviseur Xen en version 3.4.2 avec un noyau Linux-2.6.32 en dom0. Ces nœuds proposent une capacité de 2.1 uCPU et 3.5 GB de RAM pour les VM. Trois des nœuds de service exportent par NFS les images disque des VM de RUBiS et exécutent simultanément l'application de test de charge. Le dernier nœud de service exécute Plasma, configuré pour une réponse maximale en 10 secondes.
Nous avons exécuté trois instances de l'application Web RUBiS (respectivement `$A1`, `$A2` et `$A3`) en mode lecture et écriture. Chaque instance utilise sept VM, soit un total de 21 VM à administrer (cf. listing 5a). Chaque VM de `$T1` exécute le serveur Apache 2.1.12 et utilise 512 MB de mémoire. Chaque VM de `$T2` exécute le serveur d'application Tomcat 7.0.2 et utilise 1 GB de mémoire. Finalement, chaque VM de `$T3` exécute la base de données MySQL-cluster 7.1.5 en mode répliquée et utilise 1 GB de mémoire. Le répartiteur de charge utilisé par Apache est mod_jk 1.2.28. Il est utilisé pour rediriger les requêtes vers les serveurs d'application. Ces derniers sont connectés aux bases de données via Connector/J 5.1.13. Les VM sont initialement placées de manière à satisfaire leur contraintes de placement.

```
1  //Description de l'infrastructure
2  ...
3  $small = {WN[1..4], WN[5..8]};
4
5  //Description d'une instance
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};
```

(a) Description de l'environnement dans Plasma



(b) Nombre de clients simultanés lors de l'exécution de l'expérience pour chacune des instances

Reconfiguration sous pics de charge

Pour évaluer l'impact de Plasma sur les performances, nous avons réalisé, pour un même scénario, trois expérimentations : sans consolidation, avec consolidation statique et avec consolidation dynamique contrôlée par Plasma. Le scénario de test simule des montées en charge pour chaque instance de l'application Web (voir Figure 5b). Dans l'expérimentation sans consolidation, chaque VM est placée sur un nœud dédié selon les contraintes de placement. Cette expérimentation, servant de référence, nous permet d'obtenir les performances optimales des trois instances [4]. Dans l'expérimentation avec consolidation statique, chaque VM est placée de manière à satisfaire les contraintes de placement et selon sa consommation en ressource moyenne obtenue après une première exécution. Dans l'expérimentation avec consolidation dynamique, le placement des VM est contrôlé par Plasma.

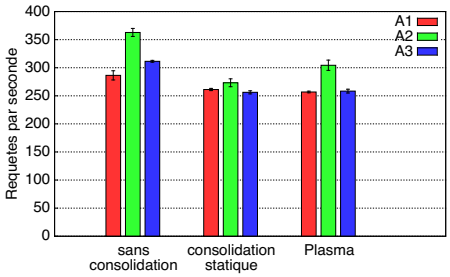


FIGURE 5: Débit moyen et erreur-type pour chaque instance après 5 répétitions des expériences.

La Figure 5 illustre le nombre moyen de requêtes traitées par seconde pour chaque instance dans le cadre des trois expérimentations. L'utilisation de la consolidation statique a réduit le débit de 17,7% alors que l'utilisation de Plasma ne l'a réduit que de 14,7%. Sans consolidation, 21 nœuds sont cependant requis pour héberger les applications, alors que la consolidation a permis d'exécuter les applications sur uniquement 8 nœuds, soit une réduction de 62%. Nous remarquons que le débit global de Plasma est 3% supérieur à celui de la consolidation statique. En effet, même si les pics de charge des scénarios n'ont pas tous lieu simultanément, plusieurs serveurs se sont retrouvés surchargés temporairement. Avec la consolidation statique, cette situation dégrade les performances des applications Web. Avec Plasma cependant, ces configurations non-viables sont corrigées par des migrations. Sur cinq répétitions des scénarios, Plasma a effectué en moyenne 12 reconfigurations pour une durée moyenne de 29 secondes. La reconfiguration la plus longue a nécessité 89 secondes (incluant le temps de calcul) et 10 migrations.

Reconfiguration depuis des événements externes

Dans cette expérimentation, la reconfiguration du placement n'est pas liée uniquement à la consommation en ressource des VM, mais également à des opérations de maintenance.
Le tableau 1 résume le scénario de l'expérimentation. À différents moments, des contraintes `ban` sont injectées (+) ou retirées (−) pour réaliser des maintenances. Lorsque la configuration courante devient

4. Le nombre de nœuds disponible ne permettant pas d'exécuter toutes les applications simultanément, chaque application a été exécutée indépendamment. Cette restriction n'a cependant pas d'impact sur les résultats.

Moment	Évènement	Reconfiguration
2'10	+ ban({WN8})	3 + 3 migrations (0'42)
4'30	+ ban({WN4})	2 + 7 migrations (1'02)
7'05	- ban({WN4})	-
11'23	+ ban({WN4})	*pas de solutions*
11'43	- ban({WN8}) + ban({WN4})	2 migrations (0'28)

TABLE 1: Événements externe se produisant durant l'expérience. '+' indique l'ajout d'un contrainte, et '-' son retrait.

non-viable, le solveur dispose alors de 10 secondes pour calculer une nouvelle configuration viable. Le nombre de migrations et le temps de reconfiguration sont détaillés dans la troisième colonne du tableau. L'ajout d'une contrainte `ban` sur un nœud impose de migrer les VM de ce nœud vers d'autres nœuds. Ces migration peut avoir des effets de bord et entraîner de nouvelles migrations. Par exemple, à 2'10, l'ajout de la contrainte `ban`({`WN8`}) entraîne la migration des 3 VM hébergées sur `WN8`, et impose la migration de 3 autres VM pour libérer des ressources et accueillir les VM de `WN8`. Une situation semblable est observable à 4'30. De telles migrations sont difficiles à détecter manuellement.
L'avant dernier événement illustre une situation sans solution : à 11'23, la contrainte `ban`({`WN4`}) est ajoutée par l'administrateur. Cependant, le solveur l'informe qu'aucun ré-agencement n'est possible. À 11'43, le retrait de la contrainte sur `WN8` par l'administrateur, permet finalement de mettre en maintenance le nœud `WN4` en entraînant la migration de 2 VM.

4.2. Passage à l'échelle de Plasma

Résoudre un RP est NP-difficile et dépend entre autre des demandes en ressource des VM, des contraintes de placement injectées et du nombre de VM et de nœuds à administrer. Nous évaluons ici l'impact de la demande uCPU et de la taille du problème sur le processus de résolution de configurations non viables, pour un centre de données simulé hébergeant des applications HA.
Le centre de données simulé est composé de 200 nœuds, logés dans 4 armoires et proposant deux classes de latence (Listing 6a). Il héberge 20 applications Web 3-tiers (`$A1` à `$A20`), dont les VMs sont dimensionnées comparablement aux classes définies par Amazon EC2 [1]. Chaque application est composée de 20 VM, architecturée selon le Listing 6b. 400 VM sont donc hébergées sur le centre de données.
Pour évaluer l'impact des contraintes de placement, nous avons évalué deux approches : (`RP-Core`) qui est le RP premier, sans contraintes de placement et (`RP-HA`) qui intègre les contraintes de placement. Le temps de migration d'une VM est estimé à une seconde par giga-octet de RAM, et le temps de lancement à dix secondes. Initialement, les VM sont placées selon leurs contraintes de placement. La demande uCPU des VM est ensuite déterminée aléatoirement entre 0 et leur maximum afin de produire des configurations non-viables. Finalement, 1% des nœuds est mis hors-ligne pour simuler des pannes matérielles.

```
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};
```

(a) Centre de données simulé. Les nœuds de `$R1` et `$R2` ont une capacité de 8 uCPU et de 32 GB de RAM. Les nœuds de `$R3` et `$R4` ont une capacité de 14 uCPU et de 48 GB de RAM.

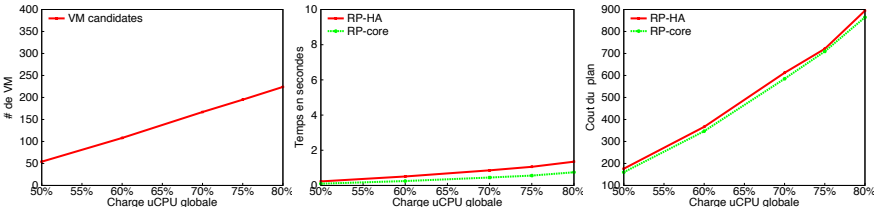
```
1 $T1 = VM[1..5];
2 $T2 = VM[6..15];
3 $T3 = VM[16..20];
4 spread($T1);
5 spread($T2);
6 spread($T3);
7 latency($T3, $medium};
```

(b) Application Web simulée. Les VM de `$T1` et `$T2` utilisent 7.5 GB de RAM et au plus 4 uCPU chacune. Les VM de `$T3` utilisent 17.1 GB de RAM et au plus 6.5 uCPU chacune.

Plasma est exécuté par une JVM Sun 1.6u21 avec 5 GB de mémoire allouée, sur un nœud bi-processeur quadri-cœur Intel Xeon-L5420 à 2.5 GHz et 32 GB de mémoire, sur lequel Plasma n'utilise qu'un cœur. Pour chacune des expérimentations, un temps limite pour calculer le meilleur plan de reconfiguration a été fixé. Trois situations sont alors possibles, le module de planification peut avoir calculé au moins une solution, prouvé qu'il n'y a pas de solutions, ou ne pas avoir pu statuer sur la faisabilité du problème.

Impact de la demande globale en uCPU

Pour cette évaluation, le module de planification est utilisé pour une demande uCPU globale des applications variant de 50% à 80% de la capacité de l'infrastructure. Le temps limite de calcul pour résoudre un RP est fixé à 2 minutes. Pour les 5 niveaux de charge testés, 100 configurations ont été résolues.



(a) Nb. de VM candidates (b) Temps de calcul (c) Coût des plans

FIGURE 6: Impact de la demande uCPU globale sur le processus de résolution.

Pour la totalité des problèmes de cette évaluation, le solveur a calculé au moins une solution. La Figure 6a montre que le nombre de VM candidates sélectionnées par le module de planification augmente avec la demande uCPU. Pour une charge de 50%, 54 VM sont sélectionnées, à comparer au 224 VM candidates lorsque la charge est de 80%. Cette augmentation est liée au nombre croissant de nœuds devenant surchargés par l'accroissement des besoins uCPU des VM.

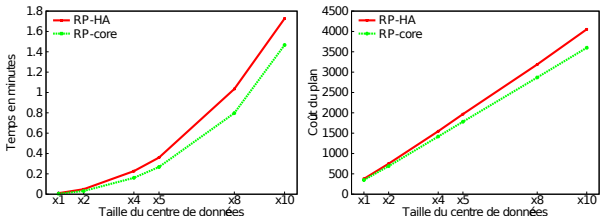
La Figure 6b présente le temps moyen de calcul de la première solution. Nous remarquons que ce temps augmente légèrement, de 0.2 à 1.4 secondes, en fonction de la charge global. Ceci s'explique par le nombre croissant de VM candidates à placer. Le temps de résolution est légèrement plus long pour l'approche `RP-HA` que `RP-core`. Ceci est dû aux algorithmes des contraintes de placement ajoutés pour résoudre les `RP-HA`. Ce temps supplémentaire peut cependant être considéré comme négligeable.

Finalement, nous observons dans la Figure 6c que le coût de la reconfiguration augmente avec le taux de charge global. En effet, comme beaucoup de VM sont candidates, un plus grand nombre de VM peut être amené à être migré pour atteindre une configuration viable. De plus, certaines migrations ont pour effet de bord, d'imposer de nouvelles migrations pour libérer des ressources sur le nœud cible.

En pratique, nous constatons que la première solution calculée est presque toujours la meilleure : sur 500 solutions calculées, seules 14 ont été améliorées ultérieurement. Il est à noter que le temps supplémentaire nécessaire pour améliorer une solution n'est intéressant que si il est inférieur à la réduction du coût apportée par la nouvelle solution. Dans nos expérimentations, 11 des 14 améliorations ont alors été rentables et 9 d'entres elles ont été calculées moins d'une seconde après la première solution. Nous pouvons en conclure qu'un temps de calcul de 3 secondes est suffisant pour cette expérimentation.

Impact de la taille du problème

Pour cette expérimentation, nous étudions l'impact de la taille du problème sur le temps de résolution. Nous avons défini six ensembles de configuration différents par leur nombre de VM et de nœuds. L'ensemble `x1` définit le centre de données simulés standard, avec 400 VM et 200 nœuds. Les ensembles `x2` à `x10` sont alors deux à dix fois plus important, en termes de nombre de VM et nœuds. La demande uCPU globale est fixée à 60%. Le temps limite de calcul pour résoudre un RP est fixé à 10 minutes.



(a) Temps de résolution (b) Coût des solutions

FIGURE 7: Impact de la taille de l'infrastructure sur le processus de résolution.

Tous les problèmes traités par cette évaluation ont été résolus. La Figure 7 montre que le temps de calcul de la première solution (Figure 7a) et son coût (Figure 7b) augmentent en fonction de la taille du problème. L'augmentation est plus rapide pour `RP-HA` que pour `RP-core` : sur l'ensemble `x10`, résoudre `RP-HA` demande 15 secondes de plus que `RP-Core` et le coût de la solution augmente de 12%. Cette différence s'explique par le grand nombre de contraintes de placement à considérer.
Nous observons finalement que l'augmentation du temps de calcul est exponentiel, en accord avec la nature NP-difficile du problème. Cette durée reste tout de même raisonnable rapportée à la taille des problèmes traités. Ainsi, avec l'ensemble `x10` et pour `RP-HA`, le solveur prend en moyenne 100 secondes pour 1) placer 1117 VM candidates sur 1980 nœuds selon leur besoins en ressource, 600 contraintes `spread`, et 200 contraintes `latency`, et 2) ordonnancer 475 actions.

5. Travaux apparentés

Consolidation dynamique

Nathuji et al. [10] présentent des algorithmes pour contrôler et coordonner différentes stratégies d'économie d'énergie, incluant la migration de VM. Bobroff et al. [4] ré-agencent les VM selon une prédiction de leur besoin en ressources. Verma et al. [14] incluent en plus un modèle de consommation énergétique des VM et des nœuds. Finalement, Yazir et al. [17] proposent une approche considérant une quantité variable de critères de placement relatifs à la consommation en ressources des VM : CPU, latence, etc.. Le calcul du placement est décomposé en tâches indépendantes, exécutées par plusieurs agents autonomes pour améliorer le passage à l'échelle. Tout ces travaux proposent des heuristiques ne pouvant être spécialisées par de nouvelles contraintes de placement comme celles relatives à la haute disponibilité.
DRS [15] est le gestionnaire de ressources de VMWare. Il fournit à l'administrateur système un outil similaire à la contrainte `ban` et une règle similaire à `spread`. Les contraintes `latency` et `fence` ne sont pas disponibles. Les administrateurs des applications ne peuvent pas déclarer leurs contraintes. De plus, les actions de migration de l'administrateur système surclassent les règles de placement et peuvent provoquer des inconsistances. L'administrateur doit donc vérifier manuellement que ses actions sont compatibles avec les règles définies. Finalement, à notre connaissance, les règles sont implantées par des heuristiques qui sont évaluées individuellement, limitant les possibilités de résolution de problèmes de reconfiguration complexes. Plasma propose une approche considérant simultanément toutes les contraintes afin de calculer une solution globalement optimisé. Hermenier et al. [7] proposent une approche reposant sur a programmation par contraintes pour placer les VM. L'algorithme embarque néanmoins une heuristique qui ne peut maintenir de nouvelles contraintes de placement telles que `spread`. Finalement, tandis que Plasma sélectionne une petite fraction de VM pour réparer une configuration non-viable, Entropy considère toutes les VM, limitant son passage à l'échelle à quelques centaines de VM et de nœuds.

Gestion d'applications n-tiers

Pradeep et al. [11] considèrent un centre de données hébergeant simultanément plusieurs applications multi-tiers. Les VM sont placées manuellement par l'administrateur système. Un gestionnaire de ressources ajuste alors dynamiquement leur distribution pour chaque tier afin de satisfaire le contrat de service des applications. Jung et al. [8] calculent le nombre de réplicas et la configuration hors-ligne. Des règles d'ordonnancement pour les VMM sont ensuite générées d'après les contrats de service afin d'adapter l'ordonnancement en direct, selon la demande en ressources des VM. Cette approche a ensuite été étendue pour générer des règles de relocation pour les VM lorsque des nœuds sont surchargés [9]. Ces travaux se focalisent sur l'adaptation structurelle des applications en fonction de leur charge. Ils ne considèrent pas les contraintes de placement liées à la haute disponibilité.

6. Conclusion et travaux futurs

La consolidation permet d'héberger plusieurs VM sur un même nœud. Les applications modernes ont cependant des besoins de passage à l'échelle et de tolérance aux pannes qui sont alors difficiles à intégrer. Nous avons présenté Plasma, un gestionnaire de consolidation dynamique configurable par des scripts, permettant à l'administrateur système et aux administrateurs des applications de décrire leurs contraintes de placement. Les scripts sont interprétés à la volée pour paramétrer un algorithme de pla-

cement premier. L'algorithme spécialisé résultant ré-arrange alors le placement des VM lorsque celles-ci n'ont plus accès à suffisamment de ressources ou lorsque des contraintes ne sont plus satisfaites. Des évaluations sur des données simulées montrent que l'introduction de contraintes de placement impacte peu le temps d'exécution de l'algorithme. L'implémentation actuelle calcule en moins de 2 minutes, des reconfigurations pour un centre de données simulé de 2000 nœuds hébergeant 4000 VM avec 800 contraintes. Sur une grappe de 8 nœuds exécutant 3 instances de l'application RUBiS sur 21 VM, la consolidation fournie par Plasma atteint 85% des performances d'une grappe de 21 nœuds.
Nos prochains travaux porterons sur l'intégration de contraintes de placement adressant d'autres problématiques ainsi que des contraintes pouvant être violées. Nous souhaitons également optimiser le passage à l'échelle de Plasma en détectant des problèmes indépendants, solvables en parallèle.

Bibliographie

1. Amazon EC2 Instance Types. `http://aws.amazon.com/ec2/instance-types/`.
2. 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 *19th ACM Symposium on Operating Systems Principles*, pages 164–177. ACM Press, October 2003.
3. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005.
4. N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. *10th IFIP/IEEE International Symposium on Integrated Network Management, 2007*, pages 119–128, May 2007.
5. C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, May 2005.
6. F. Hermenier, A. Lèbre, and J. Menaud. Cluster-wide context switch of virtualized jobs. In *4th International Workshop on Virtualization Technologies in Distributed Computing*, 2010.
7. F. Hermenier, X. Lorca, J. Menaud, G. Muller, and J. Lawall. Entropy : a consolidation manager for clusters. In *VEE '09 : 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.
8. G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *2008 International Conference on Autonomic Computing*, pages 23–32, Washington, DC, USA, 2008. IEEE Computer Society.
9. G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Middleware 2009*, pages 1–20, 2009.
10. R. Nathuji and K. Schwan. VirtualPower : Coordinated power management in virtualizaed entreprise systems. In *21st Symposium on Operating Systems Principles (SOSP)*, October 2007.
11. P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys 2007*, pages 289–302, New York, USA, 2007. ACM.
12. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
13. P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 5–14, 2006.
14. A. Verma, P. Ahuja, and A. Neogi. pMapper : power and migration cost aware application placement in virtualized systems. In *Middleware '08 : 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264, New York, USA, 2008. Springer-Verlag New York, Inc.
15. VMWare Infrastructure : Resource Management with VMWare DRS. Technical report, 2006.
16. L. Yang, I. Foster, and J. M. Schopf. Homeostatic and tendency-based CPU load predictions. In *IPDPS '03 : 17th International Symposium on Parallel and Distributed Processing*, page 42.2, 2003.
17. Y.O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 91 –98, jul. 2010.