THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. BORIS DJOMGWE TEABE

le jeudi 12 octobre 2017

Titre :

Performance et qualité de service de l'ordonnanceur dans un environnement virtualisé

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

M. GAEL THOMAS, TELECOM SUD PARIS

M. JEAN-MARC MENAUD, IMT ATLANTIQUE

Membre(s) du jury :

M. ANDRE LUC BEYLOT, INP TOULOUSE, Président

Mme VANIA MARANGOZOVA-MARTIN, UNIVERSITE GRENOBLE ALPES, Membre

A ma famille...

No one's born a computer scientist,
but with a little hard work,
and some math and science,
just about anyone can become one.
Barrack Obama.

Je tiens initialement à remercier tous les membres du jury pour le temps que vous avez consacré à l'évaluation de mon travail. Merci à mes rapporteurs Gaël Thomas et Jean-Marc Menaud à qui a été confiée la responsabilité de critiquer la forme et le fond de mon travail. Merci à Vania MARANGOZOVA-MARTIN et André-Luc BEYLOT qui ont accepté d'examiner mon travail.

Merci à mon directeur de thèse Daniel HAGIMONT et co-Directeur Alain TCHANA. Vous m'avez mis dans les conditions idéales pour réaliser cette thèse, tant sur le plan professionnel que personnel.

Un grand merci à nos admirables secrétaires Sylvie ARMENGAUD et Annabelle SAMSUNG. Vous êtes exceptionnelles dans votre travail et vous avez contribué à faciliter le mien.

Je remercie la grande équipe SEPIA ENSEEIHT, multi-culturelle et très animée dans le travail. Merci à Vlad Nitu, Gregoire TODESCI, Lavoisier WAPET et Mathieu BACOU avec qui j'ai terminé dans une bonne ambiance mon séjour au sein de l'équipe.

Je ne saurai oublier ma très grande famille qui m'a toujours soutenu dans tous mes choix. Vous constituez une grande partie de mes motivations et j'espère toujours vous faire honneur. Un merci particulier à mon père et ma mère qui ont toujours mis l'école au centre de tout.

Confrontées à l'augmentation des coûts de mise en place et de maintenance des systèmes informatiques, les entreprises se tournent vers des solutions d'externalisation telles que le Cloud Computing. Le Cloud se basent sur la virtualisation comme principale technologie permettant la mutualisation. L'utilisation de la virtualisation apporte de nombreux défis donc les principaux portent sur les performances des applications dans les machines virtuelles (VM) et la prévisibilité de ces performances.

Dans un système virtualisé, les ressources matérielles sont partagées entre toutes les VMs du système. Dans le cas du CPU, c'est l'ordonnanceur de l'hyperviseur qui se charge de le partager entre tous les processeurs virtuels (vCPU) des VMs. L'hyperviseur réalise une allocation à temps partagé du CPU entre tous les vCPUs des VMs. Chaque vCPU a accès au CPU périodiquement. Ainsi, les vCPUs des VMs n'ont pas accès de façon continue au CPU, mais plutôt discontinue. Cette discontinuité est à l'origine de nombreux problèmes sur des mécanismes tels que la gestion d'interruption et les mécanismes de synchronisation de bas niveau dans les OS invités. Dans cette thèse, nous proposons deux contributions pour répondre à ces problèmes dans la virtualisation. La première est un nouvel ordonnanceur de l'hyperviseur qui adapte dynamiquement la valeur du quantum dans l'hyperviseur en fonction du type des applications dans les VMs sur une plate-forme multi-coeurs. La seconde contribution est une nouvelle primitive de synchronisation (nommée I-Spinlock) dans l'OS invité.

Dans un Cloud fournissant un service du type IaaS, la VM est l'unité d'allocation. Le fournisseur établit un catalogue des types de VMs présentant les différentes quantités de ressources qui sont allouées à la VM vis-à-vis des différents périphériques. Ces ressources allouées à la VM correspondent à un contrat sur une qualité de service négocié par le client auprès du fournisseur. L'imprévisibilité des performances est la conséquence de l'incapacité du fournisseur à garantir cette qualité de service. Deux principales causes sont à l'origine de ce problème dans le Cloud : (i) un mauvais partage des ressources entre les différentes VMs et (ii) l'hétérogénéité des infrastructures dans les centres d'hébergement. Dans cette thèse, nous proposons deux contributions pour répondre au problème d'imprévisibilité des performances. La première contribution s'intéresse au partage de la ressource logicielle responsable de la gestion des pilotes, et propose une approche de facturation du temps CPU utilisé par cette couche logiciel aux VMs. La deuxième contribution s'intéresse à l'allocation du CPU dans les Clouds hétérogènes. Dans cette contribution, nous proposons une approche d'allocation permettant de garantir la capacité de calcul allouée à une VM quelle que soit l'hétérogénéité des CPUs dans l'infrastructure.

As a reaction to the increasing costs of setting up and maintaining IT systems, companies are turning to solutions such as Cloud Computing. Cloud computing is based on virtualization as the main technology for mutualisation. The use of virtualization brings many challenges. The main ones concern the performance of the applications in the virtual machines (VM) and the predictability of these performances.

In a virtualized system, hardware resources are shared among all VMs in the system. In the case of the CPU, it is the scheduler of the hypervisor that is in charge of sharing the CPU among all the virtual processors (vCPU) of the VMs. The hypervisor uses a time-sharing approach to allocate the CPU. Each vCPU has access to the CPU periodically. Thus, the vCPUs of the VMs do not have continuous access to the CPU, but rather discontinuous. This discontinuity is causing many problems on mechanisms such as interuption handling and low-level synchronization mechanisms in guest OSs. In this thesis, we propose two contributions to address these problems in virtualization. The first is a new hypervisor scheduler that dynamically adapts the quantum value in the hypervisor according to the type of applications in the VMs on a multi-core platform. The second contribution is a new synchronization primitive (named I-Spinlock) in the guest OS.

In a cloud providing a service of the IaaS type, the VM is the allocation unit. The provider establishes a catalogue presenting the different quantities of resources that are allocated to the VM regarding various devices. These resources allocated to the VM correspond to a contract on a quality of service negotiated by the customer with the provider. The unpredictability of performance is the consequence of the incapability of the provider to guarantee this quality of service. There are two main causes of this problem in the Cloud : (i) poor resource sharing between different VMs and (ii) heterogeneity of infrastructure in hosting centers. In this thesis, we propose two contributions to answer the problem of performance unpredictability. The first contribution focuses on the sharing of the software resource responsible for managing the drivers, and proposes to bill the CPU time used by this software layer to VMs. The second contribution focuses on the allocation of the CPU in heterogeneous clouds. In this contribution, we propose an allocation approach to guarantee the computing capacity allocated to a VM regardless of the heterogeneity of the CPUs in the infrastructure.

Table des matières

Chapitre 1

Introduction

1.1 Contexte Général

Confrontées à l'augmentation des coûts de mise en place et de maintenance des systèmes informatiques, les entreprises se tournent vers des solutions d'externalisation telles que le Cloud Computing. Le Cloud Computing est l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'intermédiaire d'un réseau, généralement internet. Ces serveurs informatiques sont déployés dans des *centres d'hébergement* et sont rendus accessibles comme des services à la demande. Plus simplement, il s'agit des clients/utilisateurs (c'est-à-dire entreprises/particuliers) qui accèdent à des ressources informatiques, reposant sur une architecture distante gérée par une tierce partie : le *fournisseur de Cloud*. En externalisant leurs services informatiques les clients visent à réduire les coûts liés aux systèmes informatiques en payant uniquement ce dont ils ont besoin. Les fournisseurs quant à eux visent à fournir une qualité de service convenue avec le client, mais également veulent réduire les ressources matérielles utilisées. Le Cloud Computing, à travers la mutualisation des ressources informatiques entraîne des économies autant pour les fournisseurs de Cloud que les clients. En général, un Cloud se caractérise par le modèle de service qu'il offre : une infrastructure dans le cas d'un IaaS (*Infrastructure As A Service*), plate-forme logicielle dans le cas d'un PaaS (*Platform As A Service*) et d'un logiciel dans le cas d'un SaaS (*Software As A Service*). Quel que soit le modèle de services proposé dans un Cloud, les fournisseurs de Cloud se basent sur la virtualisation comme principale technologie permettant la mutualisation.

La virtualisation se définit comme l'ensemble des techniques matérielles et/ou logicielles qui permettent de faire fonctionner plusieurs systèmes d'exploitation

(OS) sur un même serveur physique [44]. Cela est possible grâce à une couche d'abstraction logicielle communément appelée logiciel de virtualisation, ou hyperviseur. L'hyperviseur fournit une abstraction logicielle qui apparaît équivalent à un serveur physique. Cette abstraction logicielle s'appelle une machine virtuelle (VM), et accueille un OS invité. La virtualisation permet au fournisseur de Cloud d'allouer à chaque client un espace de travail isolé garantissant ainsi l'isolation du point de vue de la sécurité et également du point de vue des performances. La virtualisation apporte plusieurs avantages aux fournisseurs de Cloud tels que la consolidation des serveurs, la fiabilité et la disponibilité du système, la facilité de gestion des pannes et la sécurité. Malgré tous ces points positifs, la virtualisation transporte également son lot de défis tels que l'inefficacité dans le partage de ressources entre différentes VMs, des failles de sécurité et l'imprévisibilité des performances dû aux interférences entre les VMs. Dans le cadre de cette thèse, nous nous focalisons sur les problèmes d'efficacité de l'ordonnancement dans les environnements virtualisés et de l'imprévisibilité des performances dans les Clouds virtualisés.

1.2 Efficacité de l'ordonnancement

1.2.1 Motivation

Dans un système virtualisé, les ressources matérielles sont partagées entre toutes les VMs du système. Ainsi, l'hyperviseur doit multiplexer les ressources entre les différentes VMs. L'hyperviseur dispose des mécanismes pour le partage des ressources. Dans le cas du CPU, c'est l'ordonnanceur de l'hyperviseur qui se charge de le partager entre tous les processeurs virtuels (vCPU) des VMs. L'hyperviseur réalise une allocation à temps partagé du CPU entre tous les vCPUs des VMs. Chaque vCPU a accès au CPU périodiquement, pour une durée de temps connu sous le nom de quantum. Ainsi, les vCPUs des VMs n'ont pas accès de façon continue au CPU, mais plutôt discontinue. Cette discontinuité créée de gros problèmes dans les OS invités car ces derniers ont été conçus sous l'hypothèse que l'OS peut réquisitionner le CPU en cas de besoin. Mais ceci n'est pas vrai dans un système virtualisé car la VM n'a accès au CPU que lorsque l'hyperviseur lui donne accès. Ainsi, le partage du CPU entre les vCPUs a un impact sur certains mécanismes de l'OS invité tels que la gestion des interruptions et les mécanismes de synchronisation de bas niveau. La gestion des interruptions dans un OS nécessite que le noyau puisse avoir accès immédiatement à un CPU pour le traitement d'une interruption lorsqu'elle survient. Dans un environnement virtualisé, l'OS

invité est contraint d'attendre que l'hyperviseur lui donne accès à un CPU pour le traitement d'une interruption, augmentant ainsi la latence de traitement. Ceci à un impact sur la latence des périphériques d'Entrée/Sortie, par conséquent sur les performances des applications dans les VMs. En ce qui concerne les mécanismes de synchronisation, elles sont bâties sur l'hypothèse qu'un thread ayant un verrou ne peut pas être préempté avant d'avoir libéré le verrou. Mais cette hypothèse n'est plus vraie dans un environnement virtualisé car l'hyperviseur n'est pas au courant des mécanismes de synchronisation dans les OS invités. De ce fait, il peut retirer le CPU à un vCPU exécutant un thread ayant un verrou. La conséquence est une dégradation considérable des performances des applications dans les VMs.

1.2.2 Contributions

Dans cette thèse, nous proposons deux contributions pour répondre à ces défis rencontrés dans les systèmes virtualisés. La première s'implante exclusivement dans l'hyperviseur et la seconde nécessite une collaboration entre les OS invités et l'hyperviseur. Nous avons constaté qu'une bonne valeur de quantum permet de résoudre ces problèmes venant de la discontinuité. Fort de ce constat, nous proposons un nouvel ordonnanceur nommé *AQL_Sched* [1] qui est le premier ordonnanceur de l'hyperviseur qui adapte dynamiquement la valeur du quantum en fonction du type des applications dans les VMs sur une plate-forme multi-coeurs. La seconde contribution est une nouvelle primitive de synchronisation (nommée I-Spinlock) dans l'OS invité. Cette primitive est une nouvelle implémentation des tickets spinlock qui permet à un thread dans l'OS invité de n'acquérir un verrou que si et seulement si ce thread a la garanti de libérer le verrou sans que le vCPU l'exécutant ne soit préempté par l'hyperviseur. Ceci n'est possible que par une collaboration entre les OS invités et l'hyperviseur.

1.3 Imprévisibilité des performances

1.3.1 Motivation

Dans un Cloud fournissant un service du type IaaS, un client peut effectuer la réservation et l'utilisation à la demande d'une quantité de ressources sur une in-

1. *AQL_Sched* pour Aware Quantum Length Scheduler

frastructure matérielle distante. Les fournisseurs de Cloud de type IaaS utilisent la virtualisation et la VM est l'unité d'allocation. Le fournisseur établit un catalogue des types de VMs présentant les différentes configurations de VMs pouvant être réservées par les clients. La configuration d'une VM définit les quantités de ressources qui sont allouées à la VM vis-à-vis des différents périphériques : CPU, mémoire, réseau et disque. Ces ressources allouées à la VM correspondent à un contrat sur une qualité de service négocié par le client auprès du fournisseur. Le client s'attend à un respect de cette qualité de service à tout instant. L'imprévisibilité des performances est la conséquence de l'incapacité du fournisseur à garantir cette qualité de service. Il y a problème d'imprévisibilité des performances lorsque les performances d'une application s'exécutant dans une VM du Iaas varient dans le temps. En d'autres termes, le fournisseur ne garantit pas l'isolation des performances. Deux principales causes sont à l'origine de ce problème dans le Cloud : (i) un mauvais partage des ressources entre les différentes VMs et (ii) l'hétérogénéité des infrastructures dans les centres d'hébergement.

Pour ce qui est de (i), dans le cadre de cette thèse, nous nous sommes focalisés sur le partage d'une ressource logicielle qui est le composant de l'hyperviseur responsable de la gestion des pilotes des périphériques. Il s'agit d'une VM spéciale (que nous nommerons DD[2]) responsable de la gestion les périphériques d'Entrée/Sortie pour les autres VMs. Nous constatons que le temps CPU consommé par le DD est significatif et dépend de l'activité en opérations d'Entrée/Sortie des VMs. Nous constatons également que dans les ordonnanceurs des hyperviseurs, le temps CPU utilisé par le DD pour le compte des VMs n'est pas facturé à ces VMs. Cet état des choses peut conduire à l'imprévisibilité des performances si la capacité CPU allouée au DD ne permet pas de satisfaire toutes les VMs. Dans ce cas de figure, l'activité d'une VM aura un impact sur les autres.

En ce qui concerne (ii), dans un IaaS, les VMs peuvent migrer entre des machines hétérogènes à cause des mécanismes tels que la consolidation des serveurs. La quantité de ressources allouées à une VM à sa création peut être exprimée par une valeur relative (fraction de la ressource totale) ou une valeur absolue (valeur indépendante de la nature ou de la capacité totale de la ressource). L'allocation des ressources basée sur des valeurs relatives peut causer des problèmes, car la capacité de traitement de la ressource peut varier à cause d'une migration sur des machines hétérogènes. Nous constatons que : la mémoire, le disque et le réseau sont généralement alloués avec des valeurs absolues d'allocation. Mais ceci n'est pas le cas pour le CPU qui en général est alloué avec des valeurs relatives. Cette allocation du CPU basée sur des valeurs relatives conduit à un non-respect de la

2. DD correspond à Domain Driver

qualité des services quand les VMs sont migrées sur des machines hétérogènes, ce qui aboutit au problème d'imprévisibilité des performances.

1.3.2 Contributions

Dans cette thèse, nous proposons deux contributions pour répondre au problème d'imprévisibilité des performances. La première contribution s'intéresse au partage du DD. Nous proposons un système de facturation qui comptabilise le temps CPU utilisé par le DD pour chaque VM, et par la suite facture ce temps CPU aux différentes bénéficiaires dans l'ordonnanceur de l'hyperviseur. La deuxième contribution s'intéresse à l'allocation du CPU dans les Clouds hétérogènes. Nous proposons une approche d'allocation permettant de garantir la capacité de calcul allouée à une VM quelle que soit l'hétérogénéité des CPUs dans l'infrastructure. Ceci se fait en définissant une valeur absolue d'allocation basée sur une machine de référence, et nous garantissons cette allocation en cas de migration d'une VM sur des machines hétérogènes grâce à un système de translations.

1.4 Plan du document

Ce document est organisé comme suit.

— *Contexte d'étude.* Ce chapitre présente le contexte général de nos travaux de recherche. Nous présentons dans ce chapitre le contexte scientifique de cette thèse qui est le Cloud Computing et les avantages du Cloud. Par la suite, nous présentons la virtualisation qui est la principale technologie utilisée dans le Cloud Computing. Nous présentons les différents concepts liés à la virtualisation. Le chapitre se poursuit par une présentation de quelques problèmes liés à la virtualisation.

— *L'efficacité de l'ordonnancement dans les systèmes virtualisés.* Ce chapitre porte principalement sur l'ordonnancement dans les systèmes virtualisés. Dans ce chapitre, nous décrivons les problèmes et les contributions présentés dans la section 1.2.

— *L'imprévisibilité des performances dans le Cloud.* Ce chapitre porte sur l'imprévisibilité des performances dans le Cloud. Dans ce chapitre, nous décrivons les problèmes et les contributions présentés dans la section 1.3.

Nous concluons notre document au chapitre 5 et présentons quelques perspectives liées à la mise en œuvre de nos contributions.

Chapitre 2

Contexte général et concepts

Contents

Nous présentons brièvement dans ce chapitre, le contexte dans lequel s'inscrit le sujet de la thèse. Nous détaillons un certain nombre de définitions, de concepts de base et de technologies liés à nos travaux. Le chapitre commence par présenter le modèle du Cloud Computing ainsi que les grandes définitions associées. Puis, nous développons la notion de virtualisation qui est la principale technologie utilisée dans le Cloud Computing. Enfin, nous terminons par la présentation des principales problématiques rencontrées dans la virtualisation.

2.1 Cloud Computing

2.1.1 Introduction

Actuellement, nous constatons une croissance exponentielle du nombre d'infrastructures informatiques dans les petites/moyennes/grandes entreprises, d'une dizaine à une centaine, voire des milliers d'équipements. En effet, les entreprises ont besoin d'équipements (machines) pour l'hébergement de leurs applications (exemple : les serveurs web, les serveurs de base de données, les systèmes de fichiers etc). La mise en place et la gestion d'une infrastructure informatique n'est pas sans coût [24]. Les coûts engendrés peuvent être classés en quatre catégories : le coût matériel, le coût logiciel, le coût humain et enfin le coût énergétique.

— *Le coût matériel.* Le coût matériel d'une infrastructure informatique est la valeur monétaire totale des équipements la constituant. Ce coût est fonction du nombre, de l'usure et de la qualité des équipements présents dans le parc informatique.
— *Le coût logiciel.* Le coût logiciel inclut les frais d'achat des licences des logiciels (souvent facturés au nombre d'installations), les frais d'achat d'éventuelles mises à jour ou des fonctionnalités optionnelles et les frais de maintenance.
— *Le coût humain.* Le coût humain d'une infrastructure informatique renvoie au personnel en charge de son administration et de sa maintenance. Le parc informatique d'une entreprise est un environnement dynamique. Dynamique du fait de l'évolution autant sur le plan matériel (nouveaux types de matériel ajouté), que logiciel (nouvelles versions de logiciels) [32]. Cette dynamicité a pour conséquence le besoin de recruter plus de personnes.
— *Le coût énergétique.* La dernière catégorie de coût est le coût énergétique. L'énergie consommée par les centres d'hébergement se compose de l'énergie pour le fonctionnement du matériel et pour le refroidissement. Le coût

énergétique représente environ 50 à 70% des dépenses dans un centre d'hébergement [73].

Ces coûts sont intimement liés à la taille de l'infrastructure informatique. Ainsi, plus le parc informatique est grand, plus élevés sont les coûts. De plus, l'augmentation continue des besoins de calcul pour les entreprises entraîne une croissance de l'infrastructure informatique. Il en découle que la mise sur pied et la gestion d'une infrastructure informatique nécessitent un investissement important. Or, les entreprises doivent limiter cet investissement, afin de concentrer leurs ressources financières sur leurs services ou produits qui ont une forte valeur ajoutée. C'est dans ce contexte qu'intervient l'informatique dans les nuages plus connu sous le nom de Cloud Computing.

2.1.2 Définition

Le Cloud Computing est l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'intermédiaire d'un réseau, généralement internet. Ces serveurs informatiques sont déployés dans des *centres d'hébergement* et sont rendus accessibles comme des services à la demande. Plus simplement, il s'agit des clients/utilisateurs (c'est-à-dire entreprises/particuliers) qui accèdent à des ressources informatiques, reposant sur une architecture distante gérée par une tierce partie : le *fournisseur de services*. Il existe de très nombreuses définitions du Cloud Computing, insistant parfois sur les aspects conceptuels, technologiques ou encore économiques. Nous donnons ci-dessous une définition du National Institute of Standards and Technology (NIST) [1], largement acceptée par la communauté scientifique et industrielle.

> *Cloud computing is a model for enabling ubiquitous, convenient, on-deman network access to a shared pool of configurable computing resources (e.g, networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* [55]

Traduction : le Cloud Computing est un modèle de service permettant un accès partagé, à la demande, via un réseau de télécommunications, à des ressources informatiques configurables (réseaux, serveurs, stockage, applications et services), ces ressources pouvant être rapidement approvisionnées et libérées avec un effort de gestion minimale ou une intervention minime du fournisseur de services. De cette belle et longue définition, il découle cinq caractéristiques du Cloud Computing :

1. C'est l'un des plus anciens laboratoires de science dans le monde

1. *Réservation à la demande.* Le client peut, unilatéralement, réserver/libérer les ressources en fonction de ses besoins sans interaction avec le fournisseur.
2. *Accès distant.* Les services/ressources approvisionnés par le fournisseur sont accessibles à distance, et ceci à travers des équipements standards tels que des téléphones portables, des ordinateurs, des tablettes.
3. *Mutualisation des ressources.* Le fournisseur mutualise ces ressources, qu'il attribue dynamiquement aux différents clients en fonction de la demande. Cette mutualisation des ressources est la principale caractéristique du Cloud Computing.
4. *Élasticité rapide.* L'utilisateur bénéficie d'un accès rapide et souple aux ressources. Celles-ci peuvent être réservées/libérées rapidement, et parfois de manière automatisée, pour vite répondre à des variations inattendues (croissante ou décroissance de la charge).
5. *Service mesurable.* Le fournisseur doit être capable de contrôler et mesurer l'utilisation de ressources en collectant des métriques à des niveaux d'abstraction appropriés, et communiquer ces mesures aux clients. Cela permet d'établir de la transparence entre les fournisseurs et les clients, mais également d'optimiser la gestion des ressources.

Le Cloud Computing, à travers la mutualisation des ressources informatiques, entraîne des économies autant pour les fournisseurs de services que les clients. En effet, la mutualisation de la demande favorise une exploitation maximale des ressources matérielles des fournisseurs. Les clients du Cloud bénéficient de la souplesse du modèle qui permet un accès facile et quasi immédiat à des ressources sans investissement important et sans engagement à long terme [24]. De ce fait, les coûts d'exploitation associés à l'utilisation de ces services sont moindres. Également, le modèle de facturation selon la formule *pay-as-you-go* est l'une des spécificités du Cloud Computing. On parle de paiement à l'usage. La facturation à l'usage assure le client de ne payer que ce qu'il consomme réellement, et ceci sans se préoccuper des détails des coûts de fonctionnement et des opérations d'administration. La figure 2.1 liste les principaux fournisseurs de Cloud actuellement sur le marché.

2.1.3 Classifications

Selon le NIST, les Clouds peuvent être classés soit selon le service qu'ils proposent, on parle de modèles de service, soit selon la relation entre le fournisseur et les clients, on parle de modèles de déploiement. Le NIST propose trois modèles de services et quatre modèles de déploiement dans le Cloud.

FIGURE 2.1 – Quelques fournisseurs de Cloud

Modèles de service. Nous distinguons trois modèles de services [55] : Software-as-Service (SaaS), Infrastructure-as-a-Service (IaaS) et Platform-as-a-Service (PaaS).

1. *Software-as-Service (SaaS).* Ce service de Cloud consiste en l'accès et l'utilisation d'un logiciel, via Internet. Le logiciel est déjà déployé par le fournisseur et est prêt à l'utilisation. Le service est accessible via un outil client, typiquement un navigateur Web. Comme exemple, nous avons des offres telles que : Google Apps [6], Microsoft Office 365 [11] et Google Documents [8].
2. *Infrastructure-as-a-Service (IaaS).* Service qui consiste à offrir aux clients des infrastructures informatiques. Une infrastructure informatique est caractérisée par une puissance de calcul, un espace de stockage, des accès réseaux, et d'autres ressources informatiques fondamentales afin que le client soit en mesure de déployer et d'exécuter tout logiciel. Les IaaS sont généralement implantés avec des systèmes virtualisés. Les plus populaires sont : Amazon Elastic Compute Cloud [1], Google Compute Engine [7] et Windows Azure Cloud Service [20].
3. *Platform-as-a-Service (PaaS).* Service dans lequel un environnement de développement administré, hébergé et maintenu par le fournisseur est fourni au client. Ce dernier a la capacité de déployer des applications développées ou acquises, en utilisant des langages de programmation, des bibliothèques, des services et des outils supportés par le fournisseur. Le client a le contrôle sur les applications déployées et les paramètres de configuration possibles pour l'environnement d'hébergement des applications. Google App Engine [5], Amazon Elastic Beanstalk [3] et IBM SmartCloud Appliction Services [9] sont les principaux exemples.

Chacun de ces types de services représente une offre différente dans le Cloud. La figure 2.2 met en lumière la distinction entre les préoccupations adressées par le fournisseur sur les différents services en détaillant les tâche d'administration

et de gestion qui lui incombent. Par exemple dans un IaaS, le client est responsable de gérer ce qu'il met au-dessus de son infrastructure tandis que dans le cas du SaaS le fournisseur réalise les installations logicielles sur l'infrastructure et le client utilise ces logiciels. Le PaaS est à mi-chemin entre les deux autres. Le IaaS constitue la couche la plus basse avec plus de contrôle pour le client. Elle permet l'implémentation des services des couches supérieures. Dans la suite de ce document et également dans le cadre de la thèse, le IaaS constitue notre principal centre d'intérêt.

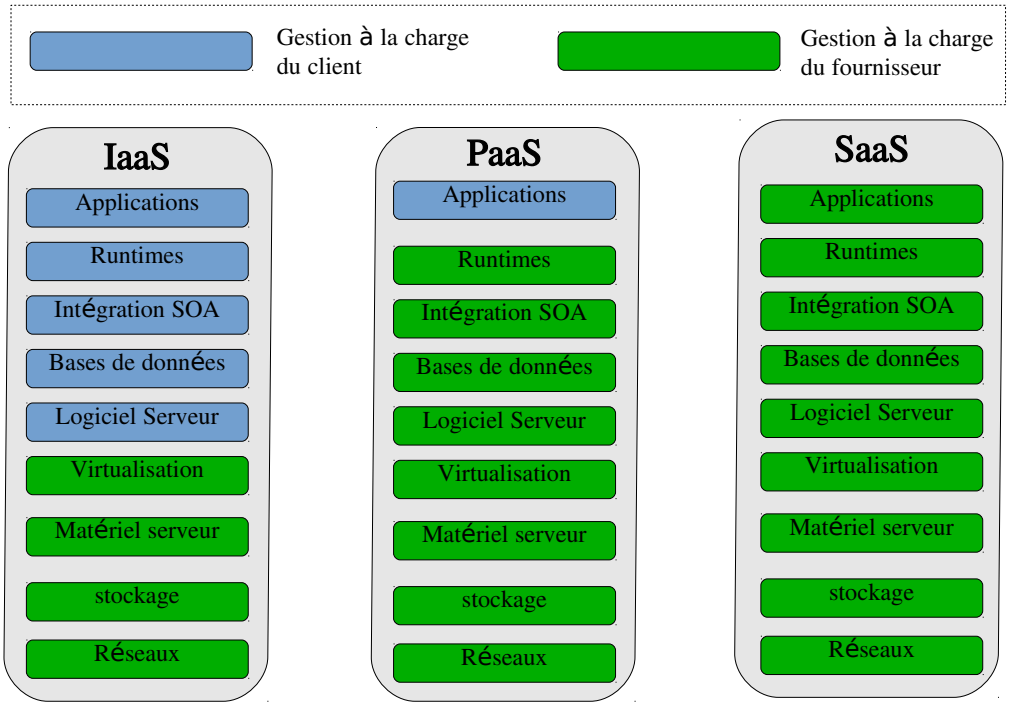

FIGURE 2.2 – Répartition des tâches d'administration des modèles de services

Modèles de déploiement. Dans ces modèles, les utilisateurs (clients) bénéficiant des services fournis par le Cloud définissent le modèle. On distingue quatre modèles (cf figure 2.3) : le Cloud privé, le Cloud communautaire, le Cloud public et enfin le Cloud hybride.

1. *Cloud privé.* Le Cloud est exploité exclusivement par une organisation, une entreprise ou administration comportant de multiples utilisateurs. Un exemple est le Cloud d'Intel [36], qui fournit des capacités de calcul à la demande pour les ingénieurs de l'entreprise.
2. *Cloud communautaire.* Le Cloud est mis en place exclusivement pour l'exploitation d'une trans-organisation (réunion de plusieurs organisations, entre-

prises ou même administrations). Comme exemple nous avons le Apps.Gov [2], qui a été mis en place par l'administration américaine pour fournir à la demande des capacités de calcul et d'hébergement pour ses entités.

3. *Cloud public.* Le Cloud est mis en place pour une utilisation commerciale, et est accessible via internet (voir figure 2.1).
4. *Cloud Hybride.* Ce Cloud est la composition ou la combinaison de deux types distincts de Clouds (privé, communautaire ou public). On retrouve généralement ce type de Cloud dans les entreprises et les organisations qui couplent les ressources de leur Cloud privé avec d'autres provenant des Clouds publics ou communautaires. Ceci dans le but de répondre à de fortes charges ne pouvant être supportées par les ressources à leur disposition dans le Cloud privé.

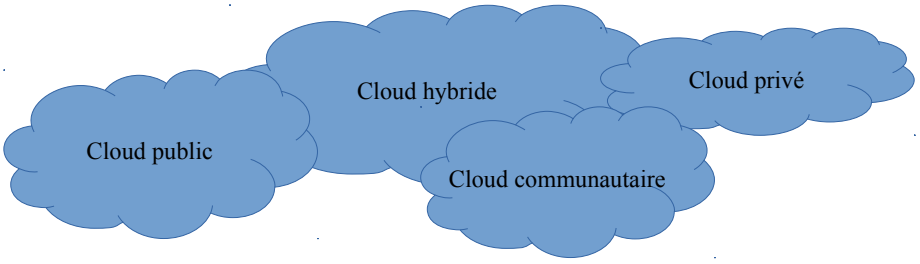

FIGURE 2.3 – Les modèles de déploiement du Cloud

2.1.4 Accord de niveau de service-SLA

La qualité de service (Quality of Service-QoS) est la capacité d'un service à répondre aux besoins des utilisateurs en respectant leurs exigences. Dans le Cloud Computing, la QoS définit le contrat entre le fournisseur et le client. Les conditions de satisfaction et les pénalités applicables au fournisseur en cas de non-respect de cette QoS sont consignées dans un document sous le nom d'accord de niveau de service (Servive Level Agreement-SLA en anglais) [50]. Le SLA contient la description et le rôle des parties impliquées, la définition de services, une période de validité du contrat et les pénalités en cas de violation [50]. Les clauses correspondent à des SLOs (Services Level Objective) que l'on peut traduire en français par "objectifs de niveau de service". Un SLO exprime un engagement à maintenir un niveau de service pendant une période donnée. Un exemple de SLO entre un fournisseur de type SaaS et ses clients peut être : 90% des requêtes traitées par le service dans une journée doivent l'être en moins de 2000ms.

Le Cloud Computing fournit au client un service qui se veut isolé et flexible. Pour ce faire il se base sur plusieurs technologies donc la plus populaire est la virtualisation. Cette dernière est la technologie la plus utilisée dans le Cloud Computing quel que soit le type de service proposé. Les principaux avantages de la virtualisation sont l'isolation qu'elle offre, l'approvisionnement dynamique et la migration facile. Dans la section suivante, nous introduisons la notion de virtualisation.

2.2 La Virtualisation

2.2.1 Introduction et définition

Durant les dernières décennies, nous constatons une omniprésence des technologies de virtualisation, notamment grâce aux nouveaux modèles de services tels que le Cloud Computing. La virtualisation des systèmes a un long passé de recherche qui remonte à 40 ans [42], avec certains de ces principes fondamentaux établis depuis les années 1970 [43].

La virtualisation se définit comme l'ensemble des techniques matérielles et/ou logicielle qui permettent de faire fonctionner plusieurs systèmes d'exploitations (OS) sur un même serveur physique. En d'autres termes, il s'agit de faire fonctionner sur une seule machine physique (PM) différents OS [44]. Cela est possible grâce à une couche d'abstraction logicielle communément appelée logiciel de virtualisation, ou hyperviseur (Virtual Machine Monitor-VMM en anglais)[2]. Le VMM fournit une abstraction logicielle qui apparaît équivalent à un serveur physique. Cette abstraction logicielle s'appelle une machine virtuelle (VM), et accueille un *OS invité* (guest). La figure 2.4 montre un exemple de virtualisation, où nous avons des réplicas de cinq systèmes physiques (exécutant cinq OS différents), qui s'exécutent sur une seule PM.

2.2.2 L'hyperviseur

L'hyperviseur est un logiciel privilégié qui fonctionne soit près ou sous un OS. Il garantit l'indépendance et l'isolation entre les différents OS invités et de

2. Dans la suite du document, l'abbréviation VMM est très souvent utilisée pour faire référence à l'hyperviseur

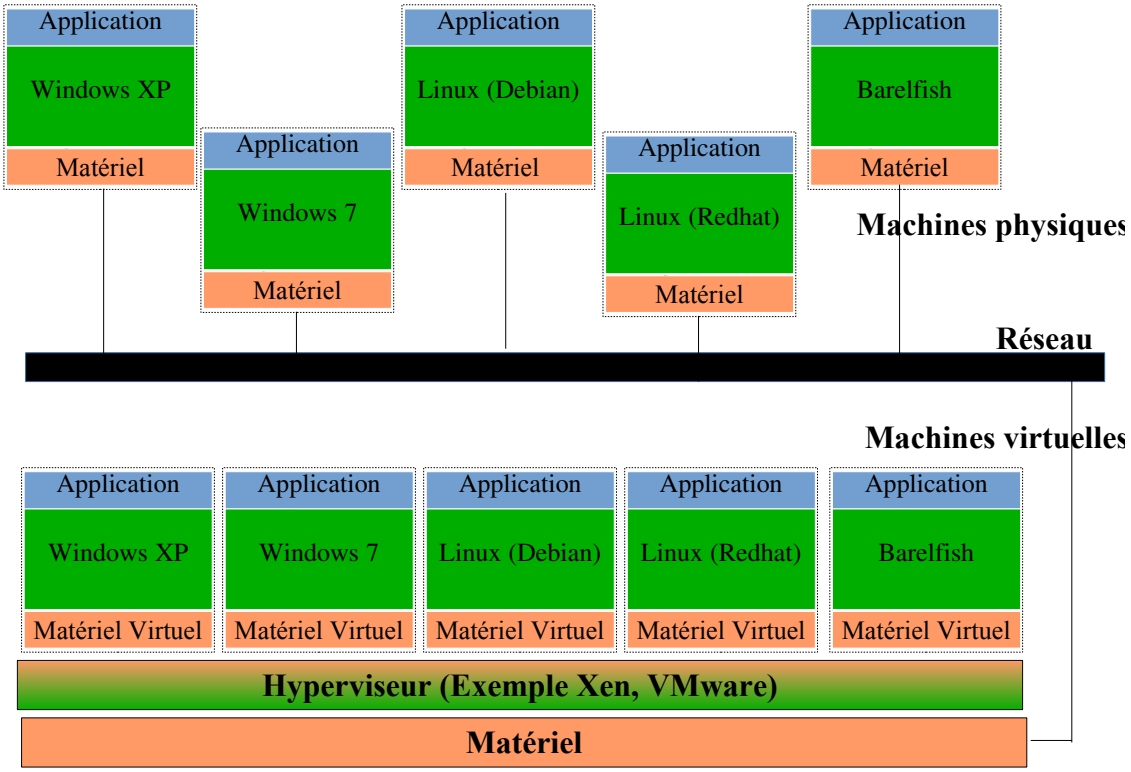

FIGURE 2.4 – Cinq machines physiques (figure du haut) et l'équivalent dans un système virtualisé (figure du bas)

ce fait fournit une meilleure sécurité pour les applications s'exécutant dans les VMs. L'hyperviseur a également pour rôle de partager et de coordonner les accès aux ressources de l'hôte physique sous-jacent entre les OS invités. Pour permettre l'exécution de plusieurs VMs, il fournit une abstraction d'une machine qui inclut un ensemble complet de ressources : processeur, mémoire, carte graphique, carte audio, disque dur, et interface réseau etc. Les hyperviseurs les plus populaires sont KVM [25], Xen [28], VMware ESXi/ESX [83] et Hyper-V [82]. Il existe deux principaux types d'hyperviseur (cf figure 2.5 Type I ou hyperviseur natif et le type II ou hyperviseur hébergé).

— *Type II ou hyperviseur hébergé.* La première technologie de virtualisation est l'hyperviseur de type II. Cet hyperviseur s'installe sur un système d'exploitation comme n'importe quel autre programme. Il se charge de virtualiser les différents périphériques nécessaires aux machines virtuelles, et transmet les demandes de ressources au système d'exploitation hôte. Avec ce type d'hyperviseur, les applications dans les OS invités souffrent d'une baisse non négligeable des performances comparées aux performances sur un système natif. Cela est dû à la présence de deux couches logicielles intermé-

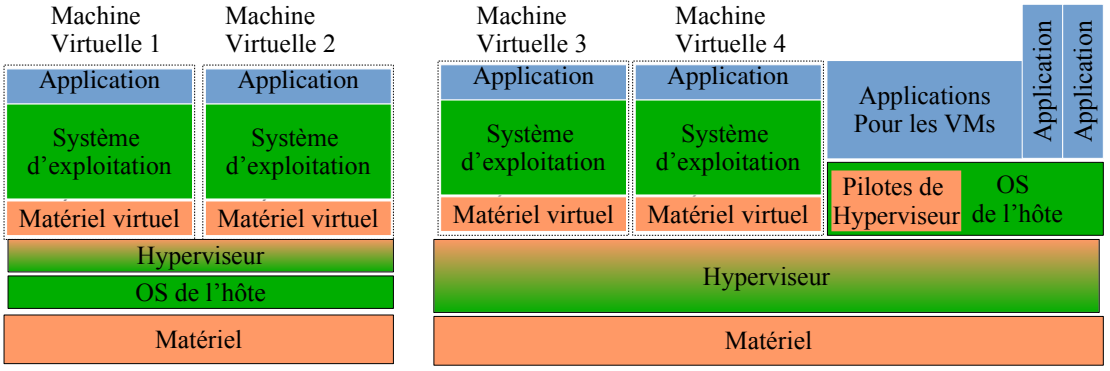

FIGURE 2.5 – Architectures des hyperviseurs de Type II (figure de gauche) et de Type I (figure de droite)

diaires : l'OS de l'hôte et l'hyperviseur. Comme exemple, nous pouvons citer : VMware Workstation, Oracle Virtualbox et Microsoft Virtual PC.

— *Type 1 ou hyperviseur natif.* Ce type d'hyperviseur s'exécute directement sur le matériel comme une couche logicielle entre le matériel et les machines virtuelles. Ainsi, les OS invités s'exécutent directement au-dessus de l'hyperviseur. L'avantage avec cette approche vient de la faible baisse de performance due à la virtualisation, l'hyperviseur se tient entre les machines virtuelles et le matériel. Cependant, ce type d'hyperviseur est difficile à déployer. Comme exemple, nous pouvons citer l'hyperviseur Xen et VMware ESX/ESXi.

Un hyperviseur de type I est allégé de manière à se « concentrer » sur la gestion des OS invités. Ceci permet de libérer le plus de ressources possible pour les VMs. Toutefois, il est possible d'exécuter uniquement un hyperviseur à la fois sur une machine. Par opposition, l'hyperviseur de type II fournit moins de ressources disponible aux VMs, mais offre l'avantage de pouvoir exécuter plusieurs hyperviseurs simultanément. La majeur partie de nos contributions sont réalisées dans l'hyperviseur Xen qui est de type I.

2.2.3 Techniques de virtualisation

Il existe trois techniques de virtualisation (cf figure 2.6) : virtualisation totale/complète, la para virtualisation et la virtualisation assistée par le matériel.

Virtualisation totale

Elle consiste en l'émulation entièrement du matériel nécessaire pour l'exécution

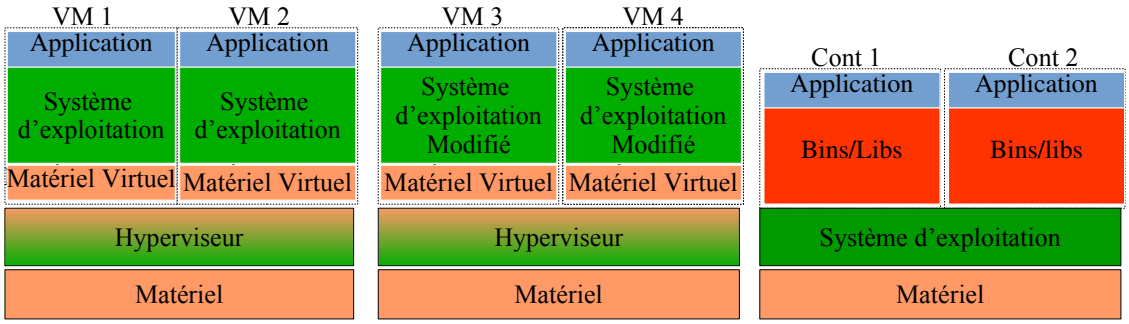

FIGURE 2.6 – Illustration de la virtualisation totale (figure de gauche), de la paravirtualisation (figure du centre) et des conteneurs (figure de droite)

d'un OS invité sans lui imposer aucune modification. La figure 2.6 de gauche illustre un exemple de virtualisation totale avec l'OS invité qui n'est pas modifié. Avec cette technique de virtualisation, l'hyperviseur émule le processeur (CPU), le stockage, la carte graphique etc. L'hyperviseur effectue les opérations nécessaires pour la traduction des instructions de l'OS invité en opérations compréhensibles et exécutables sur le support physique. Comme nous l'avons fait remarquer plus haut, le principal avantage de cette technique de virtualisation vient de la non-modification de l'OS invité. Ainsi, ces derniers ne sont pas conscients du fait qu'ils s'exécutent sur une VM. Ceci est très important lorsque les OS invités sont des systèmes d'exploitation propriétaires (exemple Windows et macOS qui ne peuvent être modifiés). Mais, cette solution de virtualisation est sujette à un overhead élevé dû au nombre important de couches logicielles lors de l'exécution de tout traitement.

Paravirtualisation

Cette technique de virtualisation a été introduite pour réduire l'overhead important observé dans l'utilisation de la virtualisation totale. Elle consiste à modifier l'OS invité afin d'y inclure des instructions spéciales nommées *hypercalls*. Ainsi, l'OS invité est modifié comme l'indique la Figure 2.6 du milieu, et l'OS invité est conscient du fait qu'il s'exécutent sur une VM. Ces dernières permettent des accès aux périphériques sans avoir recours aux traductions nécessaires dans la virtualisation totale. Cette forme de virtualisation propose des performances dans les VMs très proches des performances obtenues sur un système natif (faible overhead). Cette technique a été principalement implémentée par l'hyperviseur Xen. Toutefois, la mise en œuvre de la paravirtualisation est contraignante, car elle nécessite la modification de l'OS invité, afin d'y inclure les hypercalls.

Virtualisation assistée par le matériel

Cette technique de virtualisation utilise l'architecture du matériel pour faciliter et accroître les performances des VMs. Au fur et à mesure que la virtualisation

gagnait en popularité, les constructeurs de matériels ont intégré des nouvelles extensions dans le hard pour gérer la virtualisation. Ces extensions permettent à l'OS invité d'exécuter directement ses instructions sur la machine sans avoir besoin de traduction de la part de l'hyperviseur. Ceci implique que l'architecture matérielle que perçoit l'OS de la VM doit être identique à celle de la machine physique. Malheureusement, tous les matériels ne proposent ces extensions, et ceci limite l'adoption de la virtualisation assistée par le matériel. Avec cette forme de virtualisation, les performances des VMs sont très proches d'un OS natifs (mais sans modification de l'OS invité). Comme exemple, nous avons les technologies tels que VT-X [79] des processeurs Intel, et SVM [81] des processeurs AMD. Ces extensions proposent un ensemble d'instructions spécifiques qui facilitent la mise en place de la virtualisation sur une machine.

Les conteneurs

Depuis quelques années, la virtualisation est remise en cause du fait de la granularité des VMs souvent considérée comme lourde. En effet, une VM embarque l'OS, l'ensemble des librairies, ainsi que des applications. Une nouvelle approche a vu le jour dernièrement poussant un peu plus loin la notion de mutualisation, les conteneurs légers. Les conteneurs consistent à créer des espaces restreints, appelés *conteneurs* ou *zones*, dans lesquels des applications sont exécutées comme si le conteneur les hébergeants est un système d'exploitation à part entier. Les conteneurs d'une PM utilisent le même système d'exploitation, et c'est celui de l'hôte. Nous pouvons voir sur la Figure 2.6 de droite que l'OS hôte est partagé entre tous les conteneurs. Il faut noter que chaque conteneur à une vue des ressources disponibles : le nombre de processeurs disponibles, la quantité de mémoire disponible, la taille du disque etc. Cette technique de virtualisation propose une certaine forme d'isolation, mais moins importante que les techniques de virtualisation précédentes. Avec le partage du même système d'exploitation entre les conteneurs, ce type de virtualisation a un overhead moins important que les précédentes. Cependant, le problème de sécurité est le principal souci de cette technique de virtualisation, car tous les conteneurs partagent le même OS. Dans les techniques de virtualisation précédentes, l'hyperviseur est la couche logicielle partagée entre les VMs et il est très petit comparé à un OS, ce qui réduit les risques de faille de sécurité. Mais dans le cas des conteneurs, c'est l'OS hôte tout entier qui est partagé entre les conteneurs offrant ainsi plus de possibilités d'attaque. Des exemples d'implémentation de cette technique de virtualisation sont OpenVZ, Linux Container et Docker [87].

2.2.4 Atouts de la virtualisation

L'engouement observé pour les solutions de virtualisation n'est pas sans raison. Les principales motivations pour une ruée vers cette technologie sont les suivantes :

Consolidation des serveurs
La virtualisation rend possible la consolidation des serveurs. Cette dernière consiste à tasser les VMs sur un nombre réduit de PMs, afin d'éteindre les PMs sans VMs [90]. Elle permet de réduire la consommation énergétique. Il n'est pas rare d'obtenir un rapport de consolidation de 10 :1, c'est-à-dire 10 VMs sur 1 PM. Cela signifie que dix applications serveurs peuvent être exécutées sur une seule PMs ce qui aurait demandé dix machines physiques dans un environnement sans virtualisation. Ainsi, la consolidation des serveurs permet d'optimiser l'utilisation des ressources. La conséquence immédiate est la réduction du coût matériel pour les centres d'hébergement (réduire le nombre de machines physiques à acheter).

Test et déploiement
L'utilisation d'une VM permet un déploiement rapide en isolant l'application dans un environnement connu et contrôlé. Une VM permet de fournir à un utilisateur un environnement saint dans lequel il peut réaliser ses déploiements sans craindre que l'environnement ne soit corrompu (par exemple avec une mauvaise version des paquets installés, certains paquets endommagés etc.). Également, l'utilisation d'une VM facilite les opérations de sauvegarde qui se résument désormais à de simples copies de VMs.

Fiabilité et disponibilité du système
La virtualisation permet d'éviter des pannes du système tout entier lorsqu'il y a une erreur provenant d'une application, ou d'une VM. En effet, la virtualisation accorde un grand intérêt à l'isolation. De ce fait, des pannes survenues dans une VM n'affectent pas le comportement d'autre VM. L'utilisation de la virtualisation permet de placer les services (les tiers) d'une application dans des VMs différentes. Ainsi, si un des services est compromis, les autres ne seront pas affectés.

Sécurité
La popularité de la virtualisation vient en grande partie de l'isolation qu'elle offre. Chaque VM représente un environnement isolé, ce qui réduit le risque d'accès non autorisé et de fuite de données. Bien utilisée, l'isolation garantit qu'un code malicieux dans une VM ne peut affecter les autres VMs.

Ce travail de thèse s'inscrit essentiellement dans le contexte d'un IaaS, qui se fonde très souvent sur la virtualisation pour fournir les infrastructures au clients. Dans la section suivante nous présentons quelques problématiques rencontrées dans les systèmes virtualisés.

2.3 Problématiques dans les systèmes virtualisés

La virtualisation, comme toute technologie informatique, possède son lot de défis. Dans cette section, nous présentons ces principaux défis.

2.3.1 Problème de performance

Comme pour tout autre système informatique, avoir de bonnes performances est une exigence pour la virtualisation. L'approche intuitive pour l'évaluation des performances d'un système virtualisé est de le comparer à un système natif ayant les mêmes caractéristiques. Plus petite est la différence de performance entre les deux, meilleure est la virtualisation. Réduire, cette différence de performance est une des problématiques principales de la virtualisation, et elle peut se décliner en plusieurs défis.

Faible overhead

Comme nous l'avons dit plus haut, la virtualisation apporte des couches logicielles supplémentaires, l'hyperviseur et les OS invités. Ces couches induisent des charges supplémentaires (charge processeur et mémoire), qui ont un impact sur les performances des applications dans les VMs, c'est l'overhead de la virtualisation. Réduire cet overhead est l'un des défis majeurs de la virtualisation [70]. De nombreuses solutions autant matérielles (Intel VT pour les processeurs), que logicielles (paravirtualisation de Xen) ont été introduites pour réduire cet overhead.

Respect des SLAs

L'objectif de la virtualisation est de réaliser l'exécution simultanée de plusieurs charges sur une même machine hôte. L'hyperviseur doit permettre l'exécution simultanée d'une ou plusieurs charges, en fournissant une qualité de service. La virtualisation est le plus souvent utilisée dans le Cloud Computing. Et comme nous l'avons mentionné dans la Section 4, le fournisseur de Cloud établit un contrat de qualité de service avec le client, mentionné dans le SLA. L'hypervi-

Année	1999	2000	2001	2002	2003	2004	2005	2006	2007
Nombre de failles	1	1	2	1	9	4	10	38	34

TABLE 2.1 – Évolution du nombre de faille sur l'hyperviseur de VMWare de 1999 à 2007

seur doit partager les ressources entre toutes les VMs des clients tout en assurant la qualité de service décrite dans les SLAs des clients, tâche très difficile vu la multitude de VM que doit gérer l'hyperviseur. Il doit s'assurer qu'une VM ne puisse pas utiliser abusivement les ressources matérielles et impacter les performances des autres VMs.

Collaboration entre hyperviseur et l'OS invité
Du fait de l'isolation et de l'encapsulation, la VM est perçue comme une boîte noire dans les systèmes virtualisés. Par conséquent, l'hyperviseur n'a aucune idée de ce qui se passe dans les VMs, aucune connaissance du type, ni des besoins spécifiques des applications dans les VMs. À cause de ce manque de visibilité, il est difficile pour l'hyperviseur de faire un partage intelligent des ressources entres les VMs en fonction des besoins spécifiques des applications. L'hyperviseur n'a aucune base de référence pour faire des choix de partage intelligents, qui améliorent les performances des applications dans les VMs.

Prise en compte de d'évolution du matériel
Le matériel informatique change et se diversifie plus rapidement que les OS [29]. Actuellement, de nouvelles architectures de matériel sont produites. Ces nouveaux types de matériel nécessitent très souvent des optimisations logicielles au niveau des OS pour les utiliser au mieux. Mais dans un environnement virtualisé, les VMs ont accès à un matériel simulé par l'hyperviseur et n'ont en général aucune connaissance de l'architecture réelle de la machine hôte. De ce fait, les VMs ne peuvent tirer avantage de ces nouveaux matériels. Idéalement, l'hyperviseur doit prendre en compte les nouvelles architectures, et permettre au VMs de bénéficier des avantages.

2.3.2 Problèmes de sécurité

Avec l'essor des technologies de virtualisation, les alertes de sécurité liées à cette technologie augmentent. Le tableau 2.1 montre le nombre croissant de failles de sécurité apparaissant dans l'hyperviseur VMWare depuis 1999. La virtualisation introduit de nouvelles couches logicielles qui représentent de nouveaux domaines pouvant être victimes d'attaques. L'accès direct au matériel par ces nou-

velles couches peut causer beaucoup de dégâts. En effet, trois parties d'un système de virtualisation nécessitent une attention particulière, car elles fournissent une nouvelle aire de jeux pour les pirates informatiques.

— L'hyperviseur est certainement le plus exposé, car il établit le lien entre le matériel et les machines virtuelles. La mise à défaut de l'hyperviseur peut conduire à la compromission complète de toute l'infrastructure, et donnerait au pirate un contrôle illimité de toutes les VMs et un accès à leurs données.

— La deuxième partie sensible est la plate-forme d'administration du système de virtualisation, car elle donne un accès privilégié à toutes les instances virtuelles de l'infrastructure.

— Enfin, les extensions matérielles dédiées à la virtualisation telles que Intel VT ou AMD SVM sont également un grand danger. Elles utilisent un ensemble d'instructions spécifiques qui facilitent la mise en place de la virtualisation sur une machine. Ces extensions peuvent être exploitées pour obtenir un accès non autorisé aux ressources systèmes.

Il existe de nombreuses failles de sécurité sur les hyperviseurs actuels. Mais l'hyperviseur ayant le plus de failles est VMware [30]. Pratiquement toutes les failles sur VMWare portent sur l'élévation des privilèges. Les failles les plus critiques peuvent être exploitées par les pirates pour compromettre l'intégrité des données et la disponibilité des services.
Un exemple concret de faille dans les hyperviseurs est les canaux cachés [48]. Dans un environnement virtualisé, nous avons plusieurs VMs qui partagent les mêmes ressources. Un canal caché est une façon d'exploiter une ressource partagée comme canal de communication, à la place du mécanisme prévu. Le risque de ces canaux cachés est une communication entre VMs sans que l'hyperviseur ne soit informé.

Une autre exemple d'attaque est celui de déni de service. Dans les solutions de virtualisation, les OS invités partagent les mêmes ressources physiques, telles que le CPU, la mémoire, le disque etc. De ce fait, il est possible pour un OS invité de monopoliser les ressources du système, et laisser les autres VMs et l'hyperviseur sans accès aux ressources ; c'est une attaque par déni de service.
De tout ce qui précède, nous constatons que la virtualisation n'est pas exemptée de failles de sécurité, et l'impact d'une faille sur un système virtualisé est plus dévastateur dans un Cloud que dans un système conventionnel.

Notre travail de thèse et nos contributions s'articulent essentiellement sur l'ordonnancement dans les hyperviseurs, raison pour laquelle dans la section suivante nous introduisons quelques concepts sur l'ordonnancement dans les systèmes virtualisés.

2.4 Ordonnancement dans les hyperviseurs

Dans un environnement virtualisé, l'hyperviseur gère le matériel sous-jacent, et multiplexe les ressources matérielles. Pour la gestion du processeur (CPU), l'ordonnanceur (scheduler en anglais) de l'hyperviseur est responsable d'ordonnancer les CPUs virtuels (vCPU) sur les différents cœurs (pCPU) du CPU physique. L'OS invité considère les vCPU comme des cœurs de CPUs. Dans la VM, l'ordonnanceur de l'OS invité se charge d'ordonnancer les processus sur les vCPUs. Généralement, le partage du CPU entre vCPUs se fait suivant une approche *Round-Robin*. L'ordonnanceur de l'hyperviseur affecte un vCPU à un pCPU pour une période de temps appelée quantum avant que celui-ci soit préempté. La durée du quantum est l'unité d'allocation du CPU aux vCPUs. Dans la suite du document, pour des raisons de simplicité, nous assimilons la durée d'un quantum au quantum. De tout ce qui précède, nous pouvons déduire que l'ordonnancement dans un système virtualisé s'effectue à deux niveaux : niveau OS invité et niveau d'hyperviseur. L'ordonnanceur de l'hyperviseur a le dernier mot dans le sens où il contrôle le matériel. La figure. 2.7 présente les deux niveaux d'ordonnancement. La figure de gauche présente l'ordonnanceur dans un système natif avec un seul ordonnanceur dans l'OS, tandis que la figure de droite illustre les deux niveaux d'ordonnancement dans un système virtualisé (un ordonnanceur dans l'OS invité et un ordonnanceur dans l'hyperviseur). Le travail de cette thèse porte essentiellement sur l'ordonnancement dans l'hyperviseur. L'élaboration d'un ordonnanceur dans l'hyperviseur revient à répondre à deux questions fondamentales :

— (Q_1) quel vCPU devrait acquérir le pCPU (à un instant donné) ?

— (Q_2) pour quelle durée de temps, un vCPU peut utiliser le pCPU sans préemption (la valeur du quantum) ?

Répondre aux questions Q_1 et Q_2 permet à un ordonnanceur de donner à tous les vCPUs l'opportunité d'utiliser un pCPU, tout en évitant la famine et en assurant l'équité. La section suivante présente comment les systèmes de virtualisation les plus populaires répondent aux questions Q_1 et Q_2.

2.4.1 Ordonnancement dans Xen

Xen [28] est un hyperviseur libre, très utilisé par les fournisseurs de cloud tels que AmazXon EC2. Il propose trois ordonnanceurs : BVT (Borrowed Virtual Time), SEDF (Simple Earliest Deadline First) et Credit. L'article [35] présente une description détaillée de ces ordonnanceurs. Dans cette section, notre intérêt porte sur l'ordonnanceur Credit, car c'est celui par défaut de Xen et le plus utilisé. L'or-

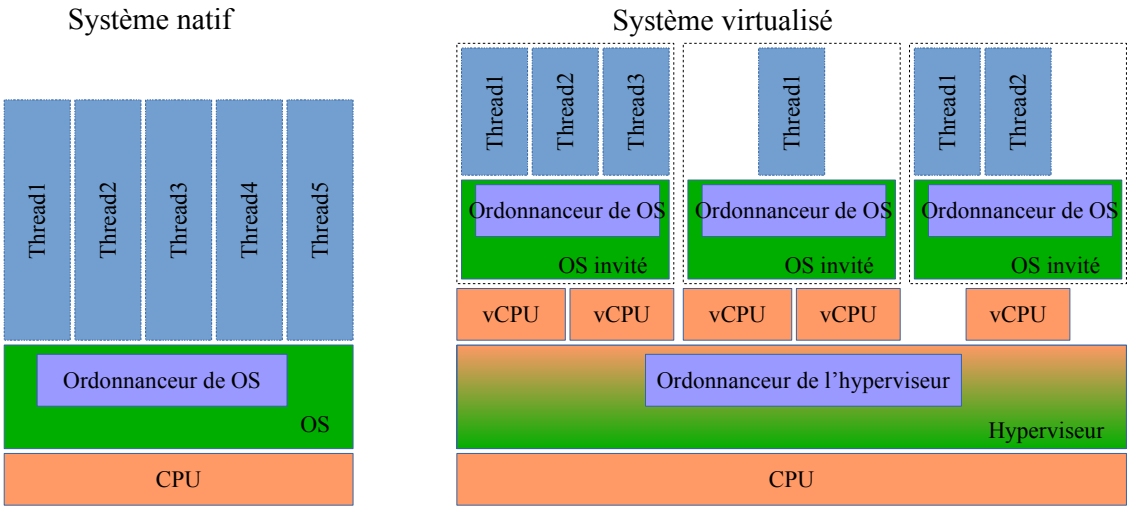

FIGURE 2.7 – Ordonnancement dans un système non-virtualisé et dans un système virtualisé.

donnanceur Credit utilise un algorithme de partage proportionnel du CPU entre les vCPUs. Chaque VM est configurée avec deux paramètres, le poids (appelé *weight*) et la capacité (appelé *cap*). Le *weight* représente la proportion de CPU à allouer à une VM en cas de partage avec d'autres VMs, tandis que le *cap* représente le pourcentage maximal de CPU que peut avoir la VM. Au démarrage d'une VM v, elle se voit affectée une quantité de crédit *remainCredit* qui est proportionnelle à la valeur du *weight*. Les vCPUs de v ont accès aux pCPUs par tranches de 30 millisecondes, qui est le quantum (réponse à la question Q_2). Chaque fois qu'un vCPU de v s'exécute sur un pCPU, (1) l'ordonnanceur convertit le temps passé par v sur le pCPU en nombre de crédits (*burntCredit*) (ceci se fait toute les 10 millisecondes). (2) Ensuite, l'ordonnanceur calcule une nouvelle valeur pour *remainCredit* en soustrayant *burntCredit* de l'ancienne valeur de *remainCredit*. Si la valeur calculée est négative, v entre dans l'état *OVER*, c'est-à-dire qu'il ne peut plus accéder à un pCPU. Sinon, v est dans l'état *UNDER*, et peut accéder à un pCPU. Après un quantum (30 millisecondes), l'ordonnanceur augmente les crédits de chaque VM en fonction du *weight* et du *cap*. Ceci permet aux VMs dans l'état *OVER* de changer d'état et d'accéder au processeur à nouveau. L'ordonnanceur Credit dispose d'une file de vCPUs par pCPU ordonnées suivant une approche *Round-Robin* (réponse à la question Q_1).

2.4.2 Ordonnancement dans KVM

KVM [25] est l'hyperviseur libre de l'OS linux. Il est intégré dans le noyau Linux depuis la version 2.6.20. Dans KVM, les vCPUs des VMs sont des threads dans l'OS de la machine hôte, de ce fait l'ordonnanceur responsable des vCPUs est celui du système Linux par défaut qui est le CFS (Completely Fair Scheduler). Dans la suite de cette section on désigne par threads les vCPUs. CFS est un ordonnanceur qui réalise un partage proportionnel du CPU entre tous les threads. CFS est une mise en oeuvre de l'algorithme WFQ (Weighted Faire Queuing) [64], dans lequel les cycles du CPU sont répartis entre les threads en fonction de leur poids (*weight*). Le *weight* est un paramètre de configuration de la VM lors de son démarrage. Par défaut tous les vCPUs des VMs ont la même valeur de *weight*. L'ordonnanceur définit un intervalle de temps fixe pendant lequel chaque thread du système doit avoir accès au CPU au moins une fois. Cet intervalle de temps est reparti entre les threads proportionnellement à leur *weight*, ce qui permet d'obtenir le quantum (réponse à la question Q_2). Lorsqu'un thread s'exécute, il accumule le *vruntime* (durée d'exécution du thread sur le pCPU divisé par son *weight*). À la fin de chaque quantum, l'ordonnanceur calcule le *vruntime* du thread actif, et choisit le thread suivant comme celui avec la plus petite valeur de *vruntime*. Dans CFS, chaque pCPU dispose d'une file de threads triés dans un arbre rouge-noir par ordre croissant de leur *vruntime*. CFS utilise un arbre rouge-noir car il permet de trier les threads par rapport à leur *vruntime*, mais également d'obtenir le thread avec le plus petit *vruntime* en $\mathcal{O}(\log n)$. Lorsque l'ordonnanceur recherche un nouveau thread à exécuter, il prend le thread le plus à gauche de l'arbre rouge-noir, ce qui correspond au thread avec la plus petite valeur de *vruntime* (réponse à la question Q_1). CFS utilise les outils tel que le *Cgroup* de linux pour limiter l'accès d'un thread à un pCPU (permet de définir un cap pour un thread).

2.4.3 L'ordonnanceur dans VMware

VMWare implémente un algorithme d'ordonnancement reposant sur le partage proportionnel et alloue le CPU aux vCPUs des VMs en fonction de leurs configurations. Dans VMware, la VM se configure avec un pourcentage (qui est l'équivalent du cap pour faire le parallèle avec l'ordonnanceur Credit de Xen) de CPU à allouer à ces vCPUs. Il est important de noter que ce pourcentage réservé n'est pas nécessairement entièrement consommé par la VM. Comme pour tous les ordonnanceurs précédents, l'ordonnanceur de VMware alloue les pCPUs aux vCPUs pour un quantum, qui correspond à 50 millisecondes (réponse à la question Q_2). À la fin du quantum, l'ordonnanceur choisit le prochain vCPU en fonction

des priorités des vCPUs. La priorité d'un vCPU est calculée en faisant le ratio du temps CPU qu'il a utilisé par le pourcentage du CPU à allouer (valeur obtenue de la configuration de la VM). L'ordonnanceur recherche le vCPU avec la plus grande priorité et lui alloue le pCPU (réponse à la question Q_1).

2.5 Synthèse

Dans ce chapitre, nous avons présenté le contexte général qui encadre nos travaux. Nous avons étudié les motivations justifiant la ruée vers Cloud Computing par les entreprises. Ces motivations s'articulent principalement sur les coûts élevés de gestion en local d'un centre hébergement, d'où la nécessité d'externalisation et de faire une utilisation au besoin. Nous avons présenté les modèles de Cloud (les modèles de services et les modèles de déploiements) présents dans la littérature et pour chaque modèle, nous avons décrit ses caractéristiques. Par la suite, nous avons présenté la virtualisaition qui est la principale technologie du Cloud. Nous avons défini les principaux concepts de la virtualisation, qui sont l'hyperviseur, la VM et l'OS invité. Et nous avons également présenté quelques défis majeurs de la virtualisation. Nombreux de ces défis sont liés aux performances des applications dans des VMs et également au respect du SLA. Dans le chapitre suivant, nous nous attaquons à ces défis de performances en nous concentrant sur l'ordonnanceur de l'hyperviseur.

Chapitre 3

L'efficacité de ordonnancement dans les systèmes virtualisés

Contents

Le chapitre commence par la présentation détaillée de quelques défis de l'ordonnancement dans les environnements virtualisés. Par la suite, nous décrivons nos deux contributions qui permettent de résoudre ces problèmes. Les descriptions de nos contributions sont toutes suivies par des évaluations. Nous concluons le chapitre par une synthèse.

3.1 Motivations

Comme nous l'avons dans le chapitre précédent, dans les systèmes virtualisés, plusieurs VMs partagent le même matériel. L'hyperviseur ordonnance les vCPUs sur les pCPUs. Lorsque le nombre de vCPU actif dépasse le nombre de pCPUs disponibles (ce qui est très souvent le cas), les pCPUs sont partagés périodiquement par tranche de temps entre vCPUs, cette tranche de temps est le quantum. Afin de réduire l'overhead dû à la commutation de contexte, les hyperviseurs utilisent généralement des grandes valeurs de quantum, de l'ordre de plusieurs dizaines de millisecondes (30 millisecondes dans Xen, et 50 millisecondes dans VMware). Cependant, ce partage de pCPUs entre les vCPUs des VMs apporte de nombreux nouveaux défis. En effet, les tâches systèmes dans les OS actuels ont été conçues sous l'hypothèse que les OS peuvent saisir les ressources du CPU à tout moment. Mais, pour les OS invités dans une VM, cette hypothèse ne peut être garantie, puisque les vCPUs des VMs partagent le même matériel et n'ont accès aux pCPUs que durant leur quantum. Rappelons qu'un système virtualisé dispose de deux niveaux d'ordonnancement, dans l'OS invité et dans l'hyperviseur. L'ordonnanceur de l'hyperviseur est celui qui attribue le matériel aux vCPUs, donc qui a le dernier mot. La figure 3.1 illustre un scénario dans lequel, malgré le fait que les thread2 de VM1, thread1 et thread3 de la VM3 sont actifs sur des vCPUs, ils n'ont pas accès aux pCPUs, car l'ordonnanceur de l'hyperviseur a choisi plutôt les vCPUs des thread3 de VM1 et thread1 de VM2. En raison de ce partage de pCPUs entre vCPUs, l'exécution d'un vCPU sur un pCPU n'est pas continue, mais discontinue. Cette discontinuité entraîne une inefficacité significative pour des mécanismes très importants de l'OS tels que les spinlock et les interruptions. Ces mécanismes reposent sur l'hypothèse de la disponibilité immédiate des CPUs chaque fois que l'OS en a besoin. Une hypothèse importante de conception des OS actuels est le fait que l'OS peut réquisitionner le CPU immédiatement pour réaliser des opérations importantes ou traiter rapidement des interruptions. Les deux sous-sections suivantes décrivent, dans les détails, les deux problèmes provenant de ces deux mécanismes (Spinlocks et interruptions) dans un système virtualisé.

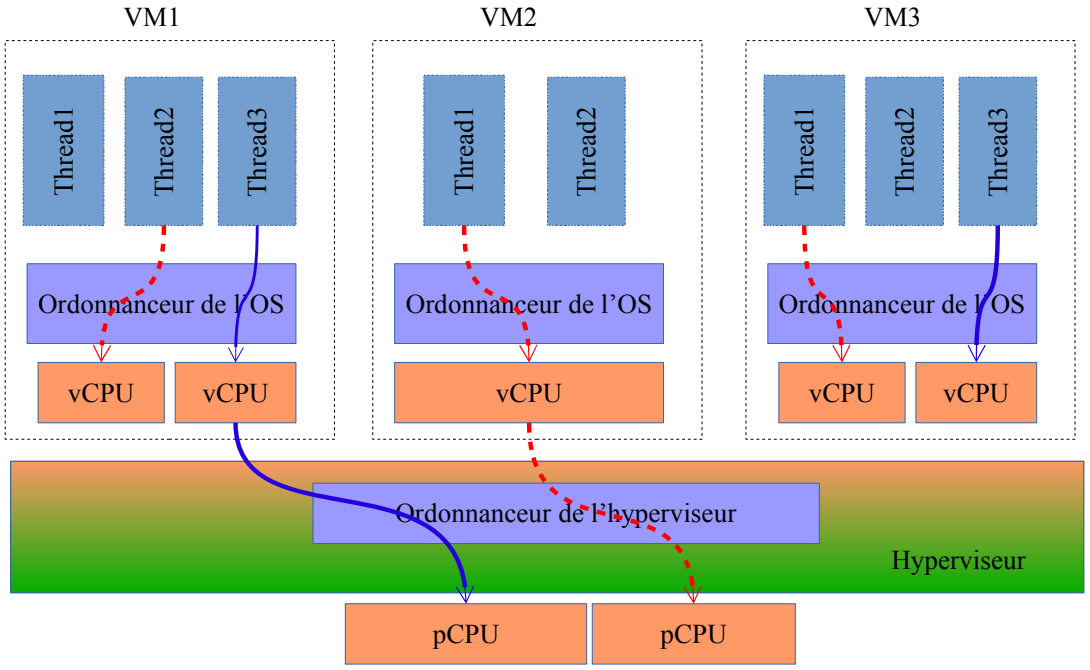

FIGURE 3.1 – Illustration de la discontinuité dans l'utilisation des pCPUs par les vCPUs des VMs

3.1.1 Spinlocks dans les systèmes virtualisés

Rappels

Les Spinlocks sont les mécanismes d'exclusion mutuelle de bas niveau dans l'OS [19] sur des CPUs multi coeurs. Ils ont deux caractéristiques principales : attente active [51] (le thread bloqué boucle en attendant de la libération du verrou) et la non-préemption des threads bloqués en attente du verrou et également du thread détenant le verrou sur leur cœur d'exécution. La non-préemption est imposée pour des besoins de performance (pas de commutation de contextes), et ceci justifie que les spinlocks sont généralement utilisés pour des verrous de courte durée. La non préemption sur un pCPU est assurée en masquant toutes les interruptions sur ce pCPU durant la section critique ou l'attente active. Les spinlocks ont une grande influence sur la sécurité et la performance de l'OS. Par conséquent, plusieurs études ont été réalisées pour leur amélioration dans l'OS Linux [54, 56]. Dans la suite de cette section, nous présentons les deux principales améliorations de spinlock dans les systèmes Linux, incorporées dans les versions du noyau $\leq$ 2.6.24 et les versions du noyau $\geq$ 2.6.25.

Spinlocks dans le noyau Linux versions $\leq$ 2.6.24. Dans ces versions, un spinlock est implémenté avec un nombre entier (ci-après dénommé *l*) qui indique si le verrou est disponible ou non. Le verrou est acquis en décrémentant et en lisant la valeur

de l (avec une instruction atomique). Si la valeur renvoyée est zéro (valeur de l), le thread acquiert le verrou. Sinon, le thread est bloqué en attente active. l prend la valeur 1 lorsque le thread libère le verrou. Pendant l'état bloqué, le thread boucle en testant la valeur de l. Il arrête de boucler lorsque l devient positif. La principale lacune de cette approche est le manque d'équité. En effet, il n'y a aucun moyen de s'assurer que le thread qui attend le plus longtemps obtient le verrou avant les autres. Cela peut conduire à la famine. Ce problème a été abordé dans les versions du noyau Linux supérieures à 2.6.24.

Spinlocks in kernel versions $\geq$ 2.6.25. Dans ces versions, les spinlocks sont implémentés avec une structure de données appelée *ticket spinlock*, qui comprend deux champs (des entiers), à savoir *head* et *queue*. Ces champs dans une structure spinlock sont respectivement initialisés à un et à zéro. Chaque fois qu'un thread tente d'acquérir le verrou, il incrémente la valeur de *queue*, puis en fait une copie (opération atomique). Cette opération s'appelle prendre un ticket et le numéro du ticket est la valeur de cette copie. L'acquisition du verrou par un thread est autorisée si son numéro de ticket est égal à *head* du spinlock. La libération du verrou est suivie de l'incrémentation de *head*. De cette façon, le verrou est acquis suivant un ordre FIFO (First In First Out).

Défis des Spinlocks dans un système virtualisé

Pour améliorer les performances des mécanismes utilisant les Spinlocks, l'OS garantit qu'un thread détenant un verrou ne peut être préempté. Cette précaution est inefficace dans un système virtualisé puisque les vCPUs sont par la suite ordonnancés sur les pCPUs, qui sont gérés par l'hyperviseur. L'hyperviseur ne fait aucune distinction entre les vCPUs qui exécutent des spinlocks et les autres. Deux problèmes peuvent résulter de cette situation, à savoir : le problème de préemption du thread détenant le verrou (Lock holder preemption) et le problème de préemption du thread bloqué en attente du verrou (Lock waiter preemption). Avant de présenter ces deux problèmes, nous présentons quelques notations. Considérons $V = \{v_1, .., v_n\}$ la liste des vCPUs qui tentent d'acquérir le même verrou. V est trié par ordre croissant des numéros de tickets de vCPU. Cela implique que v_1 est le vCPU qui détient réellement le verrou, conformément à la politique de spinlock (FIFO).

Le problème de préemption du thread détenant le verrou (LHP). Il se produit chaque fois que v_1 est préempté par l'ordonnanceur de l'hyperviseur, ce qui signifie que tout v_i, $i > 1$, consommera son quantum entier pour mener à bien l'attente active jusqu'à ce que v_1 ait de nouveau accès au pCPU et libère le verrou. Nous nommons cette attente active, *une attente active inutile*.

Le problème de préemption du thread bloqué en attente du verrou (LWP). Le problème LWP se produit à chaque fois que v_i, $1 < i < n$ est préempté et que tout les v_k, $k < i$ ont déjà utilisé le verrou. Cette situation empêche tout v_j, $j > i$, d'obtenir le verrou même s'il est disponible. Ils consommeront leur temps pour mener une attente

active inutile jusqu'à ce que v_i ait de nouveau accès au pCPU et libère le verrou. Cela s'explique par le fait que le ticket spinlock impose que le verrou soit acquis suivant un ordre FIFO.

3.1.2 La gestion des interruptions

La gestion des interruptions peut également souffrir de cette discontinuité dans l'acquisition des pCPUs par les vCPUs. Dans les systèmes non virtualisés, l'interruption est un mécanisme permettant à un périphérique d'Entrée/Sortie (E/S) d'envoyer une requête à l'OS pour un traitement immédiat. Une occurrence d'interruption invoque immédiatement le gestionnaire correspondant, en supposant la disponibilité du CPU dès que nécessaire. Le débit des périphériques E/S est affecté par la rapidité de traitement des interruptions. En outre, les IPIs (Interprocessor interrupt) pour la communication entre les pCPUs s'appuient sur le mécanisme d'interruption. En général, le traitement d'une interruption est relativement court dans les systèmes natifs. Cependant, dans les systèmes virtualisés, l'hyperviseur intercepte toute interruption réelle (interruption matérielle) et se charge de la transmettre en interruption virtuelle à un vCPU de la VM concernée. Malheureusement, si ce vCPU n'a pas immédiatement accès à un pCPU, l'interruption virtuelle ne peut être traitée par l'OS invité. L'interruption virtuelle ne sera traitée que lorsque ce vCPU aura accès au pCPU, reportant ainsi le traitement de l'interruption. Bien que le temps de traitement d'une interruption reste court dans l'OS invité, le temps d'attente pour que l'ordonnanceur de l'hyperviseur redonne accès à un pCPU au vCPU ayant reçu l'interruption augmente le temps total de traitement à des dizaines de milli secondes ou plus. Un tel délai peut réduire considérablement le débit des opérations d'Entrée/Sortie (débit d'E/S) et augmenter la latence de réponse pour les applications réalisant des opérations d'E/S (applications d'E/S).

3.2 Etat de l'art

Nombreux sont les travaux qui se sont penchés sur les questions de spinlocks et traitement des interruptions dans les environnements virtualisés.

3.2.1 Gestion des interruptions ou latence E/S.

Dans cette section, nous présentons plusieurs travaux abordant le problème de performances des E/S dans un environnement virtualisé. ils peuvent être classés en trois grandes catégories, en fonction de l'approche utilisée pour résoudre le problème.

Modification du quantum de l'ordonnanceur de l'hyperviseur
Description. Cette approche vise principalement à modifier l'ordonnanceur pour répondre au problème de gestion des interruptions. L'approche intuitive consiste à réduire le quantum de l'ordonnanceur de l'hyperviseur (microsliced [23]). Dans le cas du partage équitable du pCPU entre différents vCPUs, le temps de latence dû à une gestion tardive d'une interruption est égal au $(nombre\ de\ vCPU\ partageant\ le\ pCPU - 1) * quantum$. Ainsi, si la valeur du quantum est petite la latence d'attente avant le traitement de l'interruption le sera également. vSlicer [89] va dans la même direction en attribuant très fréquemment les pCPUs aux vCPUs exécutant des applications sensibles à la latence, réduisant ainsi le temps d'attente avant le traitement d'une interruption. vSlicer atteint son objectif en coupant le quantum de ces vCPUs en plusieurs petits quantums, et en les insérant entre les quantums des autres vCPUs n'exécutant pas d'applications sensibles à la latence.
Limite. Réduire le quantum de l'ordonnanceur de l'hyperviseur augmente le nombre de commutations de contexte, ce qui a un impact considérable sur les applications CPU intensives, mais également sur des applications utilisant beaucoup la mémoire. La commutation de contexte génère une charge CPU non négligeable lorsqu'elle est fréquente. De plus, pour les ressources partagées du matériel tel que les caches du CPU, l'augmentation du nombre de commutation des contextes rend inefficace l'utilisation de telles ressources, le cache est très souvent vidé donc pas de réutilisation des données du cache.

Meilleur choix du vCPU pour le traitement de l'interruption
Description. Cette approche préconise de choisir au mieux le vCPU pour le traitement de l'interruption afin de réduire le temps de latence. En effet, l'interruption matérielle est convertie en interruption virtuelle par l'hyperviseur qui la redirige vers un vCPU de la VM. Selon cette approche, bien choisir le vCPU vers lequel l'interruption virtuelle est dirigée permet de réduire la latence. C'est pourquoi [34] introduit vBalance qui fait le choix de diriger les interruptions virtuelles uniquement vers les vCPU actifs (vCPU utilisant un pCPU) de la VM. Ainsi, le traitement de l'interruption se fera immédiatement. Une autre solution entrant toujours dans ce cadre est vTurbo [88]. Les auteurs de ce travail proposent de dédier un ou plusieurs vCPUs d'une VM uniquement au traitement des interruptions. Ces vCPUs sont connus sous le nom de *vCPU turbo*. A chaque interruption

matérielle pour une VM, l'interruption virtuelle sera toujours dirigée vers les *vCPUs turbos*. Pour garantir une faible latence, les vCPUs turbo devront toujours avoir accès à un pCPU en cas de besoin. Pour ce faire, vTurbo propose de dédier certains pCPUs à l'utilisation exclusive des *vCPUs turbos*. Ainsi, à chaque interruption virtuelle, les *vCPUs turbos* auront toujours un pCPU disponible sur lequel ils pourront s'exécuter, car ils utilisent très peu de CPU donc les pCPUs dédiés sont pratiquement toujours disponibles.
Limite. La principale limite de vBalance [34] vient de son hypothèse de base, selon laquelle pour une VM, il est toujours possible à tout instant d'avoir au moins un de ces vCPUs en cours d'exécution, ce qui n'est pas toujours le cas. En effet, si une interruption intervient lorsqu'aucun vCPU de la VM n'est actif, alors la latence de traitement sera grande. En ce qui concerne vTurbo, la limite vient de la mobilisation des pCPUs pour le traitement des interruptions. Cette solution nécessite qu'un ou plusieurs pCPUs de la machine hôte soient dédiés à une utilisation exclusive par les *vCPUs turbos*. Cela représente un gaspillage de ressources dans le scénario où les applications dans les VMs ne réalisent pas d'opération d'E/S.

Nouveau niveau de priorité pour la gestion des interruptions
Description. Pour résoudre ce problème, l'hyperviseur Xen optimise l'ordonnanceur Credit pour le traitement des interruptions. L'ordonnanceur Credit en plus des état *OVER* et *UNDER* pour les vCPUs, ajoute un nouvel état *BOOST* [35]. Un vCPU passe dans l'état *BOOST* s'il reçoit une interruption virtuelle et, de ce fait, il verra sa priorité croître et pourra s'exécuter directement sur un pCPU. Mais un vCPU n'a le droit d'entrer dans l'état *BOOST* que s'il n'a pas utilisé tout son quantum précédent. Une telle optimisation de l'ordonnanceur permet de reproduire le comportement de la gestion des interruptions des systèmes natifs, en exécutant immédiatement un vCPU ayant reçu une interruption.
Limites. Cette approche à des limites car elle ne permet de booster un vCPU que si et seulement si, ce dernier n'a pas utilisé totalement son quantum précédent. Par conséquence, si un vCPU consomme totalement son quantum, il ne pourra entrer dans l'état *BOOST* malgré l'occurrence d'une interruption et devra attendre le prochain quantum. C'est exactement le cas lorsqu'une VM héberge deux types de charges, une charge CPU intensive et une charge réalisant des E/S. Dans ce scénario, l'application CPU intensive épuise complètement les quantum des vCPUs de la VM. De ce fait, l'ordonnanceur ne pourra plus mettre aucun vCPU de la VM dans l'état *BOOST* quand viendra une interruption virtuelle.

3.2.2 Problème des Spinlock, LWP et LHP

Plusieurs travaux de recherche ont étudié les problèmes de LHP et de LWP. L'idée fondamentale derrière toutes ces études est de détecter des situations d'attente active inutile afin de limiter les effets. Selon l'approche de détection proposée, les travaux peuvent être organisés en trois catégories.

Approches matérielles

Description. Les solutions basées sur le matériel ont été introduites dans [84]. Les auteurs ont proposé une approche de détection des vCPUs réalisant une attente active, en monitorant le nombre d'exécutions de l'instruction *pause* ("Pause-Loop Exiting" dans les processeurs Intel Xeon E5620 [47] et "Pause Filter" dans les processeurs AMD) exécutés par l'OS invité. Ils considèrent qu'un vCPU qui exécute un nombre élevé d'instructions *pause* sur un court intervalle de temps, réalise une attente active inutile. Par conséquent, l'hyperviseur peut décider de préempter de tels vCPU.

Limite. Malheureusement, même si cette fonctionnalité est en place, il reste difficile de détecter avec précision les vCPUs [23] réalisant une attente active inutile. De plus, bien que l'hyperviseur puisse détecter et préempter un tel vCPU, il est difficile de décider quel vCPU devrait obtenir le processeur [75] pour que le verrou soit libéré. En effet, en cas de préemption d'un vCPU pour attente trop longue, l'ordonnanceur de l'hyperviseur doit choisir un vCPU qui utilisera au mieux le pCPU et qui ne sera pas également bloqué en attente. [23] démontre que les solutions matérielles ont des effets mitigés : elles améliorent les performances de certaines applications tout en dégradant d'autres.

Approches Para-virtualisées

Description. Les approches para-virtualisées abordent les problèmes de LHP et LWP en modifiant les implémentations de Spinlock directement dans l'OS invité. Dans [80], les auteurs se sont concentrés sur le problème LHP. L'OS invité est modifié (de nouveaux hypercalls sont introduits) afin d'informer l'hyperviseur qu'un verrou est acquis. Ainsi, l'hyperviseur empêche sa préemption pendant la durée de l'occupation du verrou (il vise à implémenter au niveau de l'hyperviseur la caractéristique non préemptive des verrous qui est présente dans les systèmes natifs). [63] se focalise sur le problème de LWP et présente *preemptable ticket spinlocks*, une nouvelle implémentation des tickets spinlocks. Les *preemptables tickets spinlocks* permettent une acquisition du verrou par un thread sans respect de l'ordre des tickets lorsque les threads précédents sont préemptés. Ceci permet d'éviter des attentes actives inutiles. [12] a également proposé une autre implémentation de Spinlock appelée *para-virtualized ticket spinlocks* (également fondée

sur le ticket spinlock) qui traite à la fois les problèmes de LHP et LWP. Contrairement aux implémentations typiques du ticket spinlock, le para-virtualized ticket spinlock se compose de deux phases, à savoir actives et passives. Lorsqu'un vCPU essaie d'acquérir un verrou, il entre dans la phase active. Dans cette phase, le vCPU prend un numéro de ticket et commence l'attente active. Le nombre de boucle pour l'attente est limité (un paramètre configuré au moment de la compilation de l'OS). Si le vCPU n'acquiert pas le verrou pendant ces boucles, il entre dans la phase passive. Dans cette phase, le vCPU exécute une instruction (halt [1]) pour libérer le pCPU (le vCPU est suspendu). Par la suite, l'hyperviseur retire le pCPU au vCPU en attente active inutile. L'ordonnanceur exécute le vCPU de nouveau lorsque le vCPU qui l'a précédé (selon l'ordre des numéros de tickets) libère le verrou. Notons que la primitive de libération du verrou est para-virtualisée, ce qui permet à l'hyperviseur de savoir quand un verrou est libéré. Oticket [49] démontre que cette solution n'est pas scalable sur une infrastructure avec un nombre important de pCPUs et de vCPUs. Cela provient de l'utilisation d'hypercalls (pour les opérations de libération de verrou) qui sont connus pour introduire des overheads importants. Pour minimiser le nombre d'hypercalls, [49] propose d'augmenter la durée de la phase active pour certains vCPUs, ceux dont le numéro de ticket est proche de celui du vCPU qui détient le verrou.
Limitations. Premièrement, certaines de ces solutions [80, 63] ne traitent pas les deux problèmes (LHP et LWP) simultanément. Deuxièmement, presque toutes ces solutions tolèrent le LHP et LWP, et préfèrent suspendre les vCPUs qui réalisent une attente active inutile. Les vCPU suspendus ne s'exécutent pas pour la VM et, de ce fait cela a un impact sur les performances de l'application dans la VM. De plus, l'augmentation des opérations de réveil de vCPU endormi par l'instruction halt peut considérablement dégrader la performance globale des applications [37, 49]. Troisièmement, la plupart de ces solutions ne s'appliquent qu'aux VMs para-virtualisées. Les VM de type HVM ne sont pas prises en compte, car ces solutions doivent invoquer certaines fonctions implémentées dans l'hyperviseur, ce qui se fait via des hypercalls qui ne sont pas présents dans les VMs de type HVM.

Approches fondées sur l'ordonnancement dans l'hyperviseur
Description. Les solutions dans cette catégorie préconisent de prendre des mesures au niveau de l'ordonnanceur de l'hyperviseur, plutôt que dans l'OS invité. Les solutions basées sur le co-scheduling [62, 85, 57] sont situées dans cette catégorie. Elles consistent à toujours exécuter simultanément tous les vCPUs d'une VM sur les pCPUs. La préemption d'un vCPU d'une VM implique la préemption de tous ses vCPU. Ce faisant, les problèmes de LHP et LWP ne peuvent pas se produire. Une autre solution au niveau de l'hyperviseur est proposée dans [23, 86]. Elle

1. Instruction interceptée par l'hyperviseur

propose de réduire la valeur du quantum afin qu'un vCPU préempté n'attende pas longtemps avant d'avoir de nouveau accès au pCPU. De cette façon, la durée de l'attente active inutile des vCPUs sera plus court.
Limites. Bien que le co-scheduling aborde à la fois les problèmes de LHP et LWP, il présente plusieurs autres problèmes (perte d'équité et non respect des priorités) [74]. Ces problèmes sont plus pénalisants que les problèmes de LHP et LWP. En ce qui concerne les techniques de réduction de la valeur du quantum, elles augmentent le nombre de commutation de contexte , ce qui dégrade considérablement les performances pour les applications utilisant le CPU et la mémoire [23].

Nous avons introduit dans ce chapitre deux problèmes provenant du partage du CPU entre différentes VMs. Ces problèmes portent sur des mécanismes internes des OS : la gestion des interruptions et les Spinlocks. Plusieurs travaux se sont penchés sur ces problèmes et chacun à sa manière propose des solutions. Mais toutes ces solutions ont des limites (voir section 3.2) et peuvent être améliorées. Pour répondre à ces problèmes, nous pouvons, soit agir dans l'hyperviseur, soit dans l'OS invité, soit dans les deux. Nous proposons deux contributions, la première se met en place dans l'hyperviseur plus précisément sur l'ordonnancement dans l'hyperviseur et la seconde préconise une action dans l'OS invité.

— La première contribution propose un nouvel ordonnanceur de l'hyperviseur qui agit sur la valeur du quantum comme levier de solution aux problèmes. Cet ordonnanceur s'appelle *AQl_Sched*.

— La seconde contribution préconise une collaboration entre l'ordonnanceur de l'hyperviseur et l'OS invité. L'ordonnanceur de hyperviseur peut en effet transmettre aux VMs certaines informations d'ordonnancement, pour que ces dernières puissent agir efficacement et éviter les problèmes susmentionnés. Une instanciation de cette approche est une nouvelle primitive de Spinlock que nous nommons I-Spinlock.

Dans la suite du chapitre, nous présentons nos deux contributions.

3.3 Ordonnanceur à quantum variable : *AQL_Sched*

Nous avons constaté dans la section 3.2 que la réduction de la valeur du quantum est une solution aux problèmes présentés dans la section 3.1, mais que cela provoque des baisses de performances sur certaines applications. Dans cette section nous, proposons un nouvel ordonnanceur pour l'hyperviseur qui se fonde principalement sur une manipulation judicieuse de la valeur du quantum comme solution aux problèmes présentés dans la section 3.1. Cet ordonnanceur s'appelle *AQL_Sched*.

3.3.1 L'ordonnanceur *AQL_Sched*

Dans cette section, nous débutons par une présentation de l'idée de base derrière *AQL_Sched*. Ensuite, nous détaillons la conception de *AQL_Sched*. Bien que nous utilisions Xen pour le prototype, notre contribution est assez générale et s'applique à tous les systèmes de virtualisation.

3.3.1.1 Description générale

Comme indiqué dans la section 3.2, les performances dépendent fortement du quantum utilisé par l'ordonnanceur et du type (la nature) de l'application exécutée, et une bonne valeur de quantum permet de résoudre les problèmes de la section 3.1. Nous constatons qu'une valeur de quantum ne peut satisfaire toutes les applications, elle doit donc être adaptée dynamiquement en fonction de l'application. Nous définissons le type d'un vCPU à un instant donné comme celui du processus/thread utilisant le vCPU à cet instant dans la VM. **L'idée de base derrière** *AQL_Sched* **est l'ordonnancement du même type de vCPU sur un pool de pCPUs dédié, en utilisant le "meilleur" quantum (le quantum qui conduit ce type à sa meilleure performance).** Pour ce faire, l'ordonnanceur *AQL_Sched* comprend trois fonctionnalités importantes.

— *Un système de reconnaissance du type d'un vCPU (vTRS pour abréger).* Nous identifions tous les types de vCPU (au moins les plus répandus) qui pourraient être exécutés dans le Cloud. La section 3.3.1.2 présente les différents types de vCPU. Par conséquent, l'ordonnanceur *AQL_Sched* implémente un vTRS en temps réel (présenté dans la section 3.3.1.3) qui évalue périodiquement le type d'un vCPU.

— *L'identification du meilleur quantum par le calibrage.* Fondée sur des expérimentations (présentées dans la section 4.3.2), nous identifions le meilleur quantum pour chaque type d'application.

— *Le regroupement des vCPUs.* Une fois que tous les types de vCPU sont identifiés, les vCPU sont organisées en clusters en fonction de leur type. Un groupe de pCPUs est associé à chaque cluster, tout en prenant soin de garantir l'équité. Les ordonnanceurs des pCPUs qui appartiennent au même cluster sont configurés avec la même valeur de quantum, la meilleure valeur en fonction des résultats du calibrage.

Les sections suivantes détaillent la conception de notre ordonnanceur.

3.3.1.2 Les types des applications

Nous définissons trois principaux types d'application : CPU intensive, E/S intensive et concurrente. Dans cette section, nous présentons ces différents types d'applications.

Applications CPU intensives
Ce sont des applications qui utilisent intensivement le CPU, ainsi que la mémoire centrale. Nous classons les applications CPU intensives en trois sous-types en fonction de l'utilisation des caches CPU [2]. En effet, la valeur du quantum est importante pour l'utilisation des caches CPUs, car elle représente la durée pendant laquelle, un vCPU peut utiliser ses données dans le cache avant que ces données ne soient remplacées. Ainsi, une petite valeur de quantum donne très peu de temps aux vCPUs pour l'utilisation des données dans le cache. Notons également que, suivant les tailles des caches CPUs et leur hiérarchie, le dernier niveau de cache (LLC) est celui qui affecte le plus les performances des applications.
(i) *Les applications Last-level cache friendly (noté LLCF).* La taille de leur espace de travail mémoire (working set-WSS) tient dans le dernier niveau de cache (LLC). Par conséquence, elles sont très sensibles à la pollution du LLC [76, 77]. Une petite valeur de quantum conduit à une augmentation du nombre de commutations de contexte, réduisant ainsi la probabilité que les applications *LLCF* trouvent leurs données dans le LLC.
(ii) *Les applications polluantes (noté LLCO).* Leur WSS est plus grand que le LLC. Elles ne souffrent pas de la pollution du cache. Cependant, elles pourraient agir comme perturbateurs pour LLCF, selon la politique de remplacement du cache.
(iii) *Les applications low-level cache (LoLC) (noté LoLCF).* Leur WSS tient dans le LoLC (par exemple le cache L1). De telles applications sont agnostiques à la pollution du cache, car la gestion des défauts de cache du LoLC est beaucoup moins coûteuse que les défauts de cache du LLC et elles utilisent peu ou pas le LLC.

Applications E/S intensives (noté *IOInt*)
Ce sont des applications qui réalisent intensivement des opérations d'E/S, et qui sont victimes du problème de la gestion des interruptions dans les environnements virtualisés. Nous considérons à la fois les opérations sur le disque et sur le réseau. Dans la section 3.2, il est démontré dans les travaux existant qu'une grande valeur de quantum accroît la latence de traitement des interruptions et ainsi à un effet négatif sur les applications E/S. Donc, une petite valeur de quantum est préférable pour ce type d'application.

2. Les caches CPU ont un impact important sur les performances lorsqu'une mauvaise valeur de quantum est utilisée.

Applications concurrentes (noté *ConSpin*)
De telles applications sont composées de plusieurs threads qui sont en accès concurrent pour le même objet (par exemple, une structure de données) et ont donc besoin de se synchroniser. Nous nous concentrons sur des applications utilisant les Spinlocks niveau OS pour des opérations de synchronisation.

3.3.1.3 Le système de reconnaissance du type d'un vCPU (vTRS)

Le fonctionnement général
Notre ordonnanceur implémente un vTRS temps réel. Ce dernier s'appuie sur un système de monitoring qui périodiquement (toutes les 30ms, appelée période de monitoring) collecte la valeur des indicateurs nécessaires pour identifier un type de vCPU. Il prend sa décision sur le type d'un vCPU après n périodes de monitoring (fenêtre glissante). Notez qu'une petite valeur de n (par exemple 1) permet de prendre rapidement en compte les variations sporadiques du type d'un vCPU. Toutefois, cela peut avoir une incidence sur les performances de l'application qui utilise le vCPU. En effet, les variations fréquentes de type impliquent des migrations fréquentes de vCPUs entre les pCPUs (en raison du regroupement, voir la section 3.3.3). Nous constatons expérimentalement qu'une valeur de 4 pour n est idéale.

vTRS repose sur les métriques suivantes pour identifier le type d'un vCPU : le nombre de requêtes d'E/S traitées par le vCPU sur la période de monitoring (noté $IOInt\_level$), le nombre de Spinlock réalisé par la VM (noté $ConSpin\_level$), le nombre de défauts de cache LLC (noté LLC_MR_level) et le nombre de références au LLC (noté LLC_RR_level). Nous normalisons toutes les métriques afin d'avoir une unité commune, un pourcentage. Ce dernier sert de curseur qui indique la probabilité que le vCPU soit d'un type. Chaque curseur (noté xx_cur) est calculé comme suit.

curseur IOInt et ConSpin.

$$\begin{aligned} &if\ (*\_level < *\_LIMIT) \\ &\qquad *\_cur = \frac{*\_level \times 100}{*\_LIMIT} \\ &else \\ &\qquad *\_cur = 100 \end{aligned} \tag{3.1}$$

où * est soit $IOInt$, soit $ConSpin$. Pour expliquer l'équation 3.1, considérons * comme $IOInt$. $IOInt\_level$ est le nombre de requêtes d'E/S traitées au cours de

la période de monitoring précédente. $IOInt\_LIMIT$ est le seuil au-dessus duquel le vCPU est considéré comme étant à 100% du type $IOInt$.

curseur LoLCF, LLCF et LLCO. Rappelons que ces types sont des sous-types de ce que nous appelons les applications CPU intensives (voir la section 3.3.1.2), et ces types sont identifiés à partir des mêmes indicateurs (compteurs du matériel). Le calcul de leurs curseurs s'appuie sur le même ensemble de mesures et du fait qu'ils sont des sous-types d'un type, ces curseurs doivent respecter l'équation suivante :

$$LoLCF\_cur + LLCF\_cur + LLCO\_cur = 100 \tag{3.2}$$

LoLCF cursor :

$$\begin{aligned} &if\ (LLC\_RR\_level < LLC\_RR\_LIMIT) \\ &LoLCF\_cur = \frac{(LLC\_RR\_LIMIT - LLC\_RR\_level) \times 100}{LLC\_RR\_LIMIT} \\ &\quad else \\ &\quad LoLCF\_cur = 0 \end{aligned} \tag{3.3}$$

où LLC_RR_LIMIT est le nombre maximal de références aux LLC qu'un $LoLCF$ est autorisé à générer. En effet, une application $LoLCF$ fait très peu de références au LLC (pour ne pas dire aucun). Si le vCPU génère plus que LLC_RR_LIMIT, elle sera $LLCF$ ou $LLCO$.

Curseur *LLCF* :

$$\begin{aligned} &if\ (LLC\_MR\_level < LLC\_MR\_LIMIT) \\ &\quad LLCF\_cur = min(100 - LoLCF\_cur; \\ &\quad \frac{(LLC\_MR\_LIMIT - LLC\_MR\_level) \times 100}{LLC\_MR\_LIMIT}) \\ &\quad else \\ &\quad LLCF\_cur = 0 \end{aligned} \tag{3.4}$$

où LLC_MR_LIMIT est le nombre maximal de défauts de cache du LLC qu'un $LLCF$ est autorisé à générer. En effet, étant donné que $LLCF$ utilise bien le LLC, le nombre de défauts de cache du LLC qu'il pourrait générer devrait être négligeable. Au-dessus de LLC_MR_LIMIT, le vCPU est considéré comme $LLCO$ (polluante).

Curseur *LLCO* :

$$LLCO\_cur = 100 - LoLCF\_cur - LLCF\_cur \tag{3.5}$$

Une matrice de 5 lignes (une par type de curseur) et n colonnes (nombre de périodes de monitoring avant de décider du type d'un vCPU) est associée à chaque vCPU pour l'enregistrement de toutes les valeurs de métriques. À la fin de chaque période de monitoring, chaque valeur de curseur est calculée et stockée dans la dernière entrée de la ligne correspondante. La valeur moyenne de chaque ligne (noté xx_cur_avg) est ensuite calculée. Le type d'un vCPU correspond au type avec le curseur le plus élevé xx_cur_avg. Notons qu'il est difficile pour un vCPU d'avoir deux types de curseur avec les mêmes xx_cur_avg, et encore plus d'avoir xx_cur_vg égaux. Dans la section d'évaluation (section 3.3.4.1), nous montrons comment ces curseurs sont utilisés.

3.3.1.4 Le système de monitoring

Cette section présente le système de monitoring utilisé pour le calcul de $IOInt\_level$, $ConSpin\_level$, LLC_RR_level et LLC_MR_level. La construction de ce système rencontre deux défis majeurs.

— *Généricité.* Compte tenu de la diversité d'applications pouvant s'exécuter dans le cloud, vTRS ne peut s'appuyer sur aucune connaissance, sémantique, détails d'implémentation ou journaux hautement spécifiques. En outre, vTRS suppose qu'il doit fonctionner sans avoir aucun contrôle sur les VMs invitées ou les applications en cours d'exécution.

— *Overhead.* Étant donné que vTRS doit fonctionner en continu, il doit utiliser très peu de CPU.

L'utilisation de métriques de bas niveau pour inférer les types des applications dans les VMs est essentielle, car elle permet à vTRS d'identifier de manière unique les types sans nécessiter de connaissances sur les applications déployées. Le fonctionnement de vTRS repose sur des indicateurs de bas niveau.

Le système de monitoring pour *IOInt_level*

Dans l'hyperviseur Xen, l'apparition d'une opération d'E/S peut être observée au niveau de l'hyperviseur. En suivant le modèle du split-Driver [40] (utilisé par Xen), la communication entre les pilotes de périphériques d'E/S et les OS invités nécessite l'intervention à la fois de l'hyperviseur et du domaine contenant les pilotes. Par conséquent, nous proposons un système de surveillance basé sur

l'analyse des événements d'E/S dans l'hyperviseur. Chaque vCPU est associée à un compteur de requêtes E/S. Chaque fois qu'un événement sur un vCPU est lié à une requête d'E/S, le compteur d'E/S de ce vCPU est incrémenté.

Le système de monitoring pour *ConSpin_level*
Le système de monitoring est simple, car il repose sur la capacité du matériel à détecter des situations d'attente active. Par exemple, dans le processeur Intel Xeon E5620, de telles situations peuvent être piégées avec *EXIT_REASON_PAUSE_INSTRUCTION* (la fonctionnalité "Fancy", Pause Loop Exiting). Nous mettons en place un outil dans l'hyperviseur pour suivre ces situations. Pour l'implémenter sur des architectures qui ne comprennent pas *EXIT_REASON_PAUSE_INSTRUCTION*, nous proposons une seconde implémentation qui repose sur une légère modification du système d'exploitation invité. Nous intégrons un dispositif qui, durant chaque période de monitoring, compte le nombre de Spinlock opéré par une VM et met ce nombre à la disposition de l'hyperviseur par un mécanisme de mémoire partagé déjà présent dans Xen.

Le système de monitoring pour *LLC_RR_level* **et** *LLC_MR_level*
Le système de surveillance repose sur les compteurs de performance, fournis par presque tous les matériels récents. Dans le cas du système Xen, l'outil perfctr-xen [60] peut être utilisé pour collecter les défauts de cache du LLC, les références au LLC et le nombre d'instructions exécutées, nécessaires au calcul de *LLC_RR_level* et *LLC_MR_level*.

3.3.2 Le calibrage du quantum

L'une des fonctionnalités de l'ordonnanceur *AQL_Sched* est sa capacité à connaître le meilleur quantum à utiliser pour un type de vCPU donné. De même que plusieurs travaux de recherche sur ce sujet [23, 89], l'identification de la meilleure valeur de quantum requiert une phase de calibrage. Cette section présente l'ensemble des expérimentations que nous effectuons pour le calibrage.

3.3.2.1 La configuration expérimentale

Nous nous appuyons sur des benchmarks (présentés dans le tableau 3.1) soit écrits pour les besoins de cette contribution, soit tirés d'autres travaux. Nous sélectionnons ces benchmarks parce qu'ils sont représentatifs de chaque type d'application. Des expérimentations sont effectuées sur une machine HP avec un pro-

Benchmark	Description	Type
Wordpress [21]	applicaton web	*IOInt*
Kernbench [10]	compilation de Linux	*ConSpin*
[38]	parcours d'une liste chainée	*LoLCF*, *LLCF*, and *LLCO*

TABLE 3.1 – Benchmarks utilisés pour le calibrage. Chaque benchmark est représentatif de son type.

cesseur Intel (R) Core (TM) i7-3770 CPU à 3.40 GHz. Ses caractéristiques sont présentées dans le tableau 3.2. La machine exécute Ubuntu Server 12.04 virtualisé avec Xen 4.2.0. Tous les calibrages pour un type suivent le même scénario : nous exécutons une VM de base (la VM hébergeant le type d'application calibré) avec différentes configurations de VM co-localisée (afin de faire un mix d'applications co-localisées). Durant le calibrage, nous avons fait varier le nombre de vCPUs co-localisés (2 vCPU et 4 vCPU) partageant le même pCPU et la valeur du quantum. Nous utilisons quatre valeurs de quantum : 1ms, 10ms, 30ms, 60ms et 90ms. Nous avons constaté qu'une valeur de quantum supérieure à 90ms n'apporte aucun gain sur tous les types d'applications, de même qu'une valeur de quantum inférieure à 1ms. Sauf indication contraire, une VM se configure avec un seul vCPU. La section suivante présente les résultats de notre calibrage. Notons que tous ces résultats obtenus dépendent de la plate-forme.

3.3.2.2 Les résultats du calibrage

Tous les résultats présentés dans cette section (rapportés dans la figure 3.2) sont normalisés par la performance de l'application lorsqu'il est exécuté avec le quantum par défaut de Xen (30ms). Plus le graphique de performance est bas, meilleur est la performance.

IOInt. De la figure 3.2 (a), nous pouvons constater qu'un vCPU qui exécute exclusivement une charge réalisant des opérations d'E/S est insensible à la valeur du quantum. En effet, pour obtenir une faible latence, un état *BOOST* a été introduit dans Xen pour prioritiser l'ordonnancement d'un vCPU qui a été bloqué en attente d'un événement d'E/S (voir section 3.2). Malheureusement, ce mécanisme est inefficace lorsque le vCPU exécute une charge de travail hétérogène, comme on peut le voir dans la figure 3.2 (b). En effet, un vCPU est dans l'état BOOST s'il n'a pas entièrement consommé son quantum antérieur. Ce n'est pas le cas avec une charge de travail hétérogène. La figure 3.2 (b) montre que les petites valeurs de quantum sont bénéfiques pour ces charges (réalisant des opérations d'E/S).

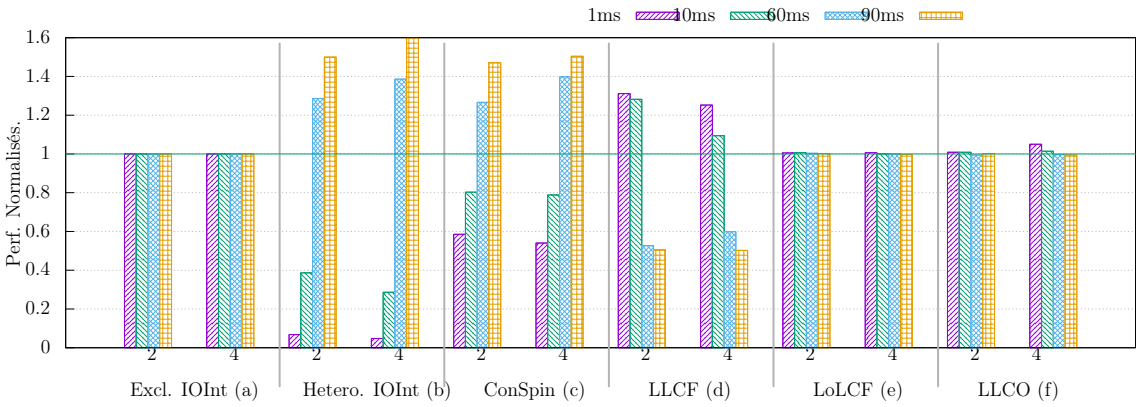

FIGURE 3.2 – Résultats du calibrage, la figure présente les résultats obtenus de notre calibrage de quantum

Mémoire centrale	8GB
L1 cache	L1_D 32 KB, L1_I 32 KB, 8-way
L2 cache	L2_U 256 KB, 8-way
LLC	8 MB, 20-way
Processor	1 Socket, 8 Cores/socket

TABLE 3.2 – Caractéristiques de notre machine expérimentale.

Selon notre discrétisation de quantum, la meilleure valeur est 1ms. Ce dernier sera donc utilisé comme quantum pour les vCPUs de type *iOInt*.

ConSpin. Pour ce calibrage, nous configurons notre VM avec 4 vCPUs. Les résultats du calibrage, présentés dans la figure 3.2 (c), montrent que le meilleur quantum pour ce type de vCPU est de 1ms.

LLCF. Le micro-benchmark [38] a été configuré pour utiliser la moitié du LLC. Fig. 3.2 (d) montre qu'un quantum élevé est meilleur pour les applications *LLCF*. La meilleure valeur de quantum que nous trouvons est 90ms.

LoLCF et LLCO. Le micro-benchmark [38] a été configuré pour utiliser 90% du cache L2 pour le calibrage de *LoLCF* tandis que plus de 100% du LLC est utilisé pour *LLCO*. La figure 3.2 (e) et (f) montrent respectivement que *LoLCF* et *LLCO* sont insensibles à la valeur du quantum. Par conséquent, ils seront utilisés pour équilibrer les clusters de vCPUs.

3.3.3 Le clustering

Après chaque invocation de vTRS, les vCPU sont organisés en clusters afin que ceux qui fonctionnent mieux avec le même quantum soient ordonnancés sur le même groupe de pCPUs. Le regroupement doit faire face à deux défis : l'équité (qui est une propriété des ordonnanceurs des hyperviseurs) et la contention LLC (sur des machines multi-sockets). Nous présentons une solution de clustering intelligent qui prend en compte ces défis. Pour ce faire, nous utilisons un algorithme de clustering à deux niveaux. Le but du premier niveau de l'algorithme est de distribuer équitablement les vCPUs sur les sockets (ensemble de pCPU) tout en évitant autant que possible la co-localisation de pollueurs (ci-après dénommés "polluant") et les vCPU sensibles à pollution du LLC (ci-après dénommés "non-polluant"). L'algorithme du deuxième niveau fonctionne à la granularité d'un socket. Tout d'abord, il organise les vCPU par cluster en fonction de leur compatibilité de quantum (voir ci-dessous). Deuxièmement, il associe équitablement un ensemble de pCPU à chaque cluster. Le reste de la section décrit les deux algorithmes.

Algorithm 1 Premier niveau de clustering.

Entrée :
totVCPUs : nombre total de vCPUs dans le système
totSockets : nombre total de sockets dans le système
Begin

polluant=∅
non-polluant=∅
classer les vCPUs tel que ceux qui appartiennent à la même VM se suivent
for each vCPU v_i **do**
    if $max(LLCF\_cur\_avg, LLCO\_cur\_avg, LoLCF\_cur\_avg) = LLCF\_cur\_avg$ **then**
        polluant=polluant∪$\{v_i\}$
    else
        non-polluant=non-polluant∪$\{v_i\}$
    end if
end for
classer les vCPUs non-polluant tel que les vCPUs *LoLCF* soient les premiers
n=$\frac{totVCPUs}{totSockets}$
for each *socket* s_i **do**
    auxSet=(polluant !=∅) ?polluant :non-polluant
    choisir les n premiers vCPUs de auxSet et les affecter au socket s_i
    appliquer Algorithm 2 au socket s_i
end for

End

Au premier niveau (algorithme 1), les vCPU sont organisés en deux groupes ("polluant" et "non-polluant") en fonction de leur intensité de pollution du LLC (lignes 4-10). Les vCPUs qui font partie de la liste des polluants sont *LLCO* (évidemment) et, *IOInt* et *ConSpin* dont le curseur de *LLCO* est grand (disons plus de 50 %). Dans ce cas, ils sont notés $IOInt^{+}$ et $ConSpin^{+}$. En ce qui concerne la liste des non-polluants, nous avons *LLCF* et *LoLCF* (évidemment), et *IOInt* et *ConSpin* (noté $IOInt^{-}$ et $ConSpin^{-}$) qui ne font pas partie de la liste des polluants. À la suite de cette première étape, les vCPUs polluants sont répartis équitablement entre les sockets (lignes 12-17). Afin de minimiser l'accès à la mémoire distante qui peut entraîner une dégradation des performances sous les architectures NUMA, notre algorithme empêche autant que possible que des vCPUs qui appartiennent à la même VM soient sur différents sockets. Ceci est obtenu en classant des vCPUs par VM avant de les affecter aux sockets (ligne 3). À des fins d'équilibrage, le socket qui héberge les derniers vCPUs polluants pourrait également recevoir des vCPUs "non-polluant" (ligne 15) lorsque le nombre de vCPU polluant n'est pas un multiple de n. En mettant les vCPUs *LoLCF* au début de la liste "non-polluante" (ligne 11), nous minimisons la probabilité de placer les vCPUs *LLCF* avec les vCPUs polluants.

Le second niveau de clustering (algorithme 2) fonctionne à la granularité du socket. Il organise les vCPU en fonction de l'affinité de quantum plutôt que des types de vCPU. En fait, à partir des résultats du calibrage, nous faisons deux observations : (1) certains types distincts atteignent leurs meilleures performances avec le même quantum (*IOInt* et *ConSin* par exemple), et (2) *LoLCF* et *LLCO* les types qui sont insensibles au quantum. À partir de ces observations, nous définissons la notion de compatibilité de quantum (*QLC* pour abréger) comme suit : un ensemble de vCPUs C, est q_QLC si tous ces vCPUs atteignent leurs meilleures performances avec le même quantum q . Par exemple, $\{IOInt, ConSpin\}$ sont $1ms\_QLC$. Par conséquent, l'algorithme de cluster fonctionne comme suit. Tout d'abord, tous les vCPUs (sauf *LoLCF* et *LLCO*) sont organisés en n clusters (lignes 2-7), n étant le nombre de quantum distinct de notre calibrage. *LoLCF* et *LLCO* sont utilisés pour équilibrer les clusters (ligne 10). Par la suite, les pools de pCPU sont construits de telle sorte que l'équité soit respectée (lignes 11-29). En affectant les pCPUs aux clusters en fonction de leur nombre de vCPUs. Au cours de cette phase, certains pCPUs (moins de n) peuvent se voir affectés des vCPUs appartenant à des clusters distincts (ligne 20). Ces vCPUs sont affecté s à un cluster avec la valeur de quantum par défaut (le quantum par défaut est utilisé par l'ordonnanceur). Enfin, l'algorithme reconfigure chaque ordonnanceur de pCPU de sorte que le quantum approprié soit utilisé (lignes 30-34). Notons que l'ordonnancement dans un cluster est assuré par l'ordonnanceur Credit de Xen, qui est censé être juste.

La figure 3.3 illustre nos algorithmes de clustering. Nous considérons une machine à quatre sockets, chaque socket ayant 4 pCPUs. Un socket est dédié au dom0 (le domaine privilégié). La machine exécute 12 $IOInt^{+}$, 7 $ConSpin^{-}$, 17 $LLCF$, et 12 $LLCO$ (un total de 48 vCPUs). Par conséquent, l'équité est respectée si chaque pCPU exécute près de 4 vCPUs. Comme le montre la figure 3.3, chaque socket reçoit exactement 16 vCPUs à la fin du premier algorithme. Avec le deuxième algorithme, 6 clusters sont formés à la fin de son exécution. Expliquons ce résultat en mettant l'accent sur ce qui se passe dans le premier et le troisième sockets. Tous les vCPUs dans le premier socket sont $1ms\_QLC$ ($IOInt$ nécessite 1ms alors que $LLCO$ est insensible à la valeur du quantum), formant ainsi un cluster unique. En ce qui concerne le troisième socket, 2 clusters ont d'abord été formés : C_4^{90} (pour les 9 $LLCF$) et C_5^1 (pour tous les 7 $ConSpin^{-}$). Sachant que l'attribution de pCPUs aux clusters doit assurer l'équité (4 vCPU par pCPU), ceci n'a pas été possible pour C_4^{90} et C_5^1. Par conséquent, un vCPU et trois vCPU ont été supprimés de C_4^{90} et C_5^1 afin de former le dernier cluster C_6^{30}. Étant donné que ce dernier contient des vCPUs qui ne sont pas QLC, il est configuré pour utiliser le quantum par défaut.

3.3.4 L'évaluation de AQL_Sched

Cette section présente les résultats de l'évaluation de notre prototype implémenté dans Xen. L'évaluation couvre les aspects suivants : l'exactitude de vTRS, l'efficacité du prototype et enfin l'overhead du prototype. Par défaut, le contexte expérimental est identique à l'environnement présenté dans la section 3.3.2.1. Notons qu'en pratique [23, 89, 88], les vCPUs du domaine privilégié (dom0) sont fixés sur des pCPUs qui leur sont dédiés. Par conséquent, ils ne sont pas pris en compte par notre ordonnanceur. Sauf notification contraire, tous les résultats sont normalisés sur les performances avec l'ordonnanceur de Xen par défaut. Une valeur normalisée inférieure à 1 signifie que l'application se comporte mieux avec Xms de quantum que 30ms de quantum.

3.3.4.1 Efficacité de vTRS

Les premières expérimentations évaluent vTRS en temps réel et valident la précision des résultats du calibrage.

Les benchmarks

Ces évaluations ont été réalisées en utilisant des benchmarks de référence : SPECweb2009 [18], SPECmail2009 [16] et SPECcpu2006 [15], qui sont respectivement

Algorithm 2 Deuxième niveau de clustering (granularité du socket).

Input :
totVCPUs : nombre total de vCPUs sur le socket
totPCPUs : nombre total de pCPUs sur le socket
Begin

i=0 //pour indexer les clusters (C_i^q)
for each quantum length q found by calibration **do**
    i++
    C_i^q =all vCPUs which are *q_QLC*
    exclure les vCPUs *LoLCF* et *LLCO* de C_i^q
    $\text{pCPUsPool}_i = \emptyset$
end for
n=i
$\text{pCPUsPool}_{n+1} = \emptyset$
utiliser les vCPUs *LoLCF* et *LLCO* pour équilibrer les clusters (C_*^*)
k=$\frac{totVCPUs}{totPCPUs}$ //nombre de vCPUs par pCPU
i=1
for each pCPU p **do**
    if k $\leq$ sizeOf(C_i^q) **then**
        S={select k unlabelled vCPUs from C_i^q}
        pCPUsPool_i=pCPUsPool_i∪{p}
    else
        S={select sizeOf(C_i^q) unlabelled vCPUs from C_i^q}
        if i<n **then**
            S=S∪{select (k-sizeOf(C_i^q)) unlabelled vCPUs from
$C_{i+1}^q, .., C_{i+j}^q$}
            les vCPUs S's sont retirés de leur clusteur initial et assignés au cluster C_{n+1}^{dq} (dq est le quantum par défaut (30ms)).
            pCPUsPool_{n+1}=pCPUsPool_{n+1}∪{p}
            i=i+j
        else
            pCPUsPool_i=pCPUsPool_i∪{p}
        end if
    end if
    marque les vCPUs de S's //Ils ont déjà été traités
end for
for each cluster C_i^q, including C_{n+1}^{dq} **do**
    for each pCPU p in pCPUsPool_i **do**
        configurer l'ordonnanceur de p's avec le quantum approrié pour le cluster's
    end for
end for
End

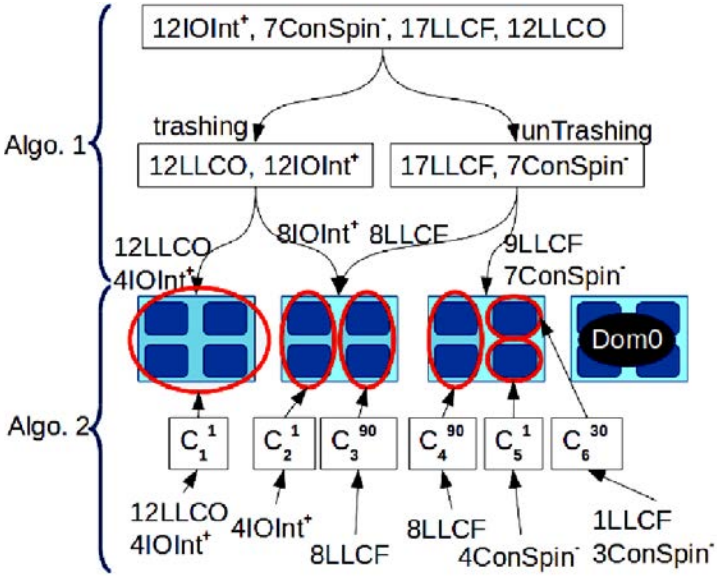

FIGURE 3.3 – Une illustration de notre solution de clustering à 2 niveaux.

des implémentations d'un service web, d'un serveur de messagerie d'entreprise et d'un ensemble d'application CPU intensives. La performance s'exprime en terme de latence moyenne des requêtes pour le premier, le temps moyen requis pour gérer une opération de messagerie pour la seconde et le temps d'exécution des programmes pour le dernier. Nous utilisons également PARSEC [31], un ensemble de programmes multi-threads, pour évaluer les applications nécessitant les Spinlocks de l'OS. Les performances du benchmark PARSEC s'expriment en temps d'exécution.

Les résultats

La figure 3.4 présente les résultats pour 5 applications représentatives, 50 valeurs de métriques $*\_cur\_avg$ utilisées par vTRS pour déduire le type d'un vCPU. Nous définissons un type d'application comme le type ayant la courbe la plus élevée la plupart du temps comparé aux autres (ceci est représenté dans la figure par les lignes rouges). Nous pouvons voir que vTRS identifie efficacement chaque type. Par exemple, SPECweb2009 est identifié comme *IOInt*, qui est son comportement connu. Le tableau 3.3 résume chaque type d'application en fonction des résultats de vTRS. Notons que pour *LLCF*, *LoLCF* et *LLCO*, ces résultats dépendent de notre environnement expérimental. En effet, une application qui a été identifiée comme *LLCO* dans notre environnement peut être identifiée comme *LLCF* sur une machine avec un LLC plus grand.

En ce qui concerne la précision du calibrage, la figure 3.5 montre les performances normalisées de chaque application lors de l'exécution avec différentes valeurs de quantum. L'environnement expérimental et la procédure d'expérimen-

IOInt	SPECweb2009, SPECmail2009
ConSpin	bodytrack, blackscholes, canneal, dedup, facesim, ferret, fluidanimate freqmine, raytrace, streamcluster
LLCF	astar, Xatanbmck, bzip2 gcc, omnetp
LoLCF	hmmer, gobmk, perlbench sjeng ; h264ref
LLCO	mcf, libquantum

TABLE 3.3 – Reconnaissance des types des applications.

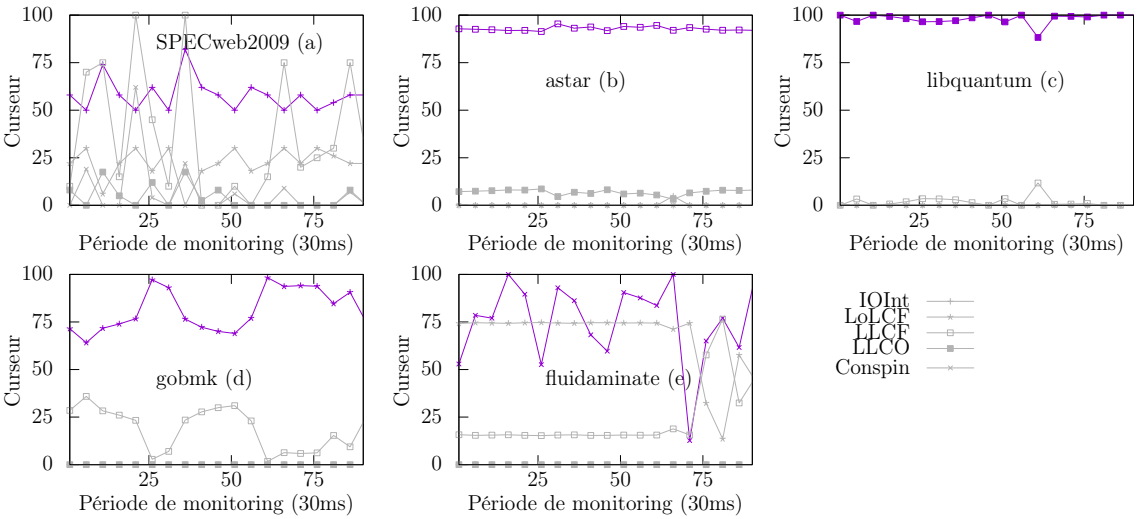

FIGURE 3.4 – Reconnaissance en temps réel avec vTRS.

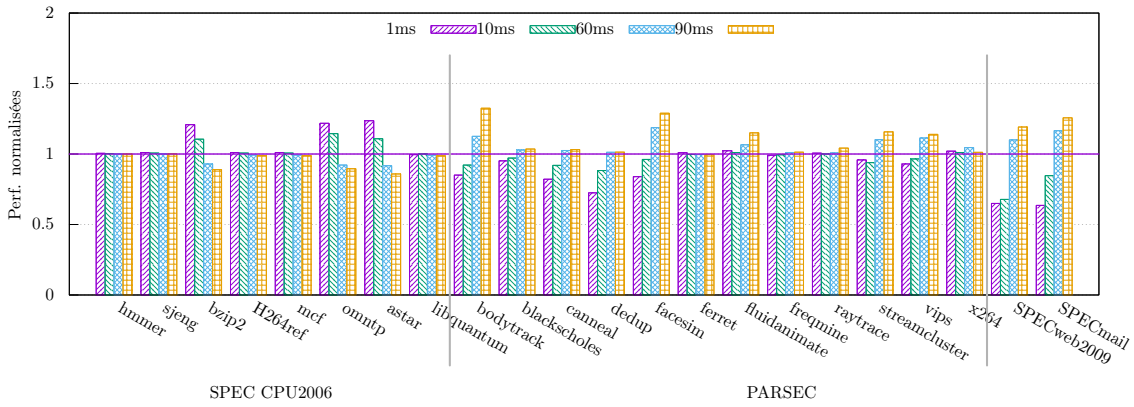

FIGURE 3.5 – Efficacité de vTRS et la précision du calibrage.

Scenarios	Colocated types	Applications
S_1	$5ConSpin$, $5LLCF$, $6LoLCF$	fluidanimate, bzip2, hmmer
S_2	$5IOInt$, $5LLCF$ $6LLCO$	SpecWeb2009, bzip2, libquantum
S_3	$5LLCF$, $5LLCO$, $6LoLCF$	bzip2, libquantum, hmmer
S_4	$4IOInt$, $4ConSpin$, $4LLCF$ $4LLCO$	SPECweb2009, facesim, bzip2 hmmer
S_5	$4IOInt$, $4ConSpin$, $4LLCF$ $2LLCO$, $2LoLCF$	SPECweb2009, facesim, bzip2, hmmer, libquantum

TABLE 3.4 – Les différents scénarios de co-localisation.

tation sont les mêmes que dans la section 3.3.2.1, mais nous bloquons l'évaluation à 4 vCPUs partageant un pCPU. Nous constatons que très souvent chaque application obtient sa meilleure performance lorsque la valeur du quantum est la meilleure pour le type détecté par vTRS. Par exemple, les meilleures performances de SPECweb2009 et SPECmail (qui sont détectés comme $IOInt$) sont obtenues lorsque le quantum est de 1ms. Rappelons que 1ms correspond au quantum obtenu après calibrage pour les applications $IOInt$. Les applications telles que dedup, fluidanimate et streamcluster sont détectées comme application $ConSpin$ et ont leurs meilleures performances avec 1ms comme quantum (valeur de quantum déterminée comme la plus appropriée pour ce type d'application d'après notre calibrage). Et enfin omntp, bzip2 et astar sont détectés comme de type $LLCF$ et la figure 3.5 montre que 90ms comme valeur de quantum fournit les meilleures performances pour ces applications, comme le dit notre calibrage.

Scenarios	Clusters	Applications	#pCPUs
S_1	C_1^1	$5ConSpin, 3LoLCF$	2
	C_2^{90}	$5LLCF, 3LoLCF$	2
S_2	C_1^1	$5IOInt, 3LLCO$	2
	C_2^{90}	$5LLCF, 3LLCO$	2
S_3	C_1^{90}	tous	tous
S_4	C_1^1	$4IOInt, 4ConSpin$	2
	C_2^{90}	$4LLCF, 4LLCO$	2
S_5	C_1^1	$4IOInt, 4ConSpin$	2
	C_2^{90}	$4LLCF, 2LLCO, 2LoLCF$	2

TABLE 3.5 – Clustering obtenu avec chaque scénario de co-localisation présenté dans le tableau 3.4.

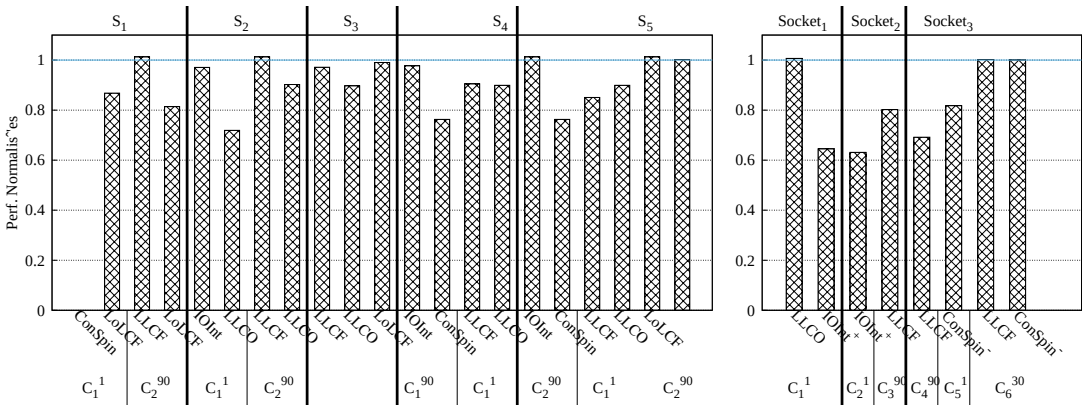

FIGURE 3.6 – Efficacité de notre prototype de AQL_Sched.

3.3.4.2 Efficacité de *AQL_Sched*

Évaluation sur une machine à une socket.
Nous évaluons d'abord l'efficacité du prototype avec des cas d'utilisation simples qui correspondent à différents scénarios d'application de co-localisation (présentés dans le tableau 3.4). Chaque scénario exécute 16 vCPUs sur 4 pCPUs, ce qui entraîne 4 vCPUs par pCPU pour l'équité. Le tableau 3.5 montre pour chaque scénario le clustering des vCPUs durant l'évaluation. La figure 3.6 de gauche présente les performances de chaque application pour chaque scénario. Nous pouvons constater, à l'exception des applications *LoLCF* et *LLCO* (qui sont quantum agnostique), que notre prototype surpasse l'ordonnanceur de Xen par défaut (jusqu'à 20% d'amélioration).

Évaluation sur une machine à 4 sockets.
Nous évaluons également notre prototype sur le cas d'utilisation complexe présenté dans la section 3.3.3. La machine expérimentale pour cette évaluation est équipée d'un processeur Intel Xeon E5-4603 (composé de 4 sockets). Pour cette expérimentation spécifique, nous nous sommes basés sur les benchmarks utilisés pour le calibrage qui sont décrits dans la section 3.3.3. Les résultats de l'évaluation sont présentés dans la figure 3.6 de droite. Rappelons que les clusters générés dans ce scénario sont présentés dans la figure 3.3. Comme indiqué ci-dessus, la pire performance obtenue avec notre prototype est identique à celle de Xen natif. Concentrons-nous sur les performances des *LLCF*, qui ne sont pas les mêmes dans les clusters C_3^{90}, C_4^{90} et C_6^{30}. Puisque C_6^{30} utilise le quantum par défaut, *LLCF* a des performances moins bonnes dans ce cluster. En ce qui concerne les *LLCF* dans C_3^{90}, ils partagent le LLC de leur socket avec les vCPUs $IOInt^{+}$, qui sont des pollueurs (dans le cluster par défaut). Ceci explique la baisse de performance de *LLCF* en C_3^{90} par rapport à C_4^{90} (qui n'héberge pas de pollueur). En outre, cela montre les avantages du système de clustering.

3.3.4.3 La comparaison avec d'autres solutions

Nous comparons notre solution à 3 autres solutions :

— vTurbo [88] : il réserve un pool de pCPUs pour l'ordonnancement des vCPUs *IOInt*, en utilisant une petite valeur de quantum (voir la section 3.2).

— vSlicer [89] : il utilise une petite valeur de quantum pour l'ordonnancement des vCPUs *IOInt*. Par rapport à vTurbo, vSlicer ne dédie pas de pool pCPU pour l'ordonnancement exclusif de vCPU *iOInt*.

— Microsliced [23] : il utilise une petite valeur de quantum pour ordonnancer tous les types de vCPUs.

Ces solutions n'implémentent aucun vTRS en temps réel. Par conséquent, nous configurons manuellement chaque solution afin d'obtenir ses meilleures performances. Nous décidons d'utiliser 1ms comme quantum pour les solutions vTurbo et Microsliced. Les évaluations sont basées sur le scénario S_5 décrit dans le tableau 3.5. La figure 3.7 présente les résultats de l'évaluation. Ces derniers sont normalisés par rapport à la performance obtenue avec l'ordonnanceur de Xen par défaut. Nous pouvons constater que notre prototype fournit au pire des cas les mêmes performances que les autres solutions. En résumé, aucune des autres solutions ne fournit de meilleures performances pour tous les types d'applications. *AQL_Sched* est le premier algorithme qui adapte le quantum au comportement de l'application, surpassant ainsi les solutions existantes.

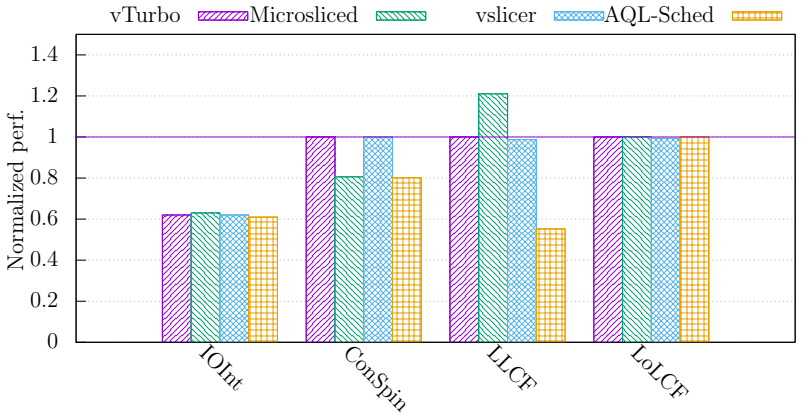

FIGURE 3.7 – Résultat de la comparaison de AQL_Sched avec les solutions existantes.

3.3.4.4 Overhead de *AQL_Sched*

Le système de monitoring
Le monitoring des opérations d'E/S est réalisé en analysant les événements dans l'hyperviseur. Cette tâche ne génère pas d'overhead puisque les mécanismes requis existent déjà dans l'hyperviseur. En ce qui concerne les systèmes de surveillance qui dépendent des compteurs matériels, nous n'observons pas d'overhead, comme l'ont également rapportés [61].

Le systèmes de reconnaissance de type et de clustering. La complexité des deux systèmes est $\mathcal{O}(max(m,n))$ où m et n sont respectivement le nombre de pCPUs et le nombre de vCPUs. Sachant que les deux valeurs sont en général inférieures à cent, (nous sommes dans le contexte des systèmes virtualisés, pas des systèmes natifs avec des milliers de tâches), $\mathcal{O}(max(m,n))$ est négligeable.

3.3.5 Synthèse

Nous venons de présenter *AQL_Sched*, le premier ordonnanceur de l'hyperviseur qui adapte dynamiquement la valeur du quantum en fonction du type des applications dans les VMs sur une plate-forme multi-coeurs. Pour ce faire, *AQL_Sched* détermine par un calibrage, la meilleure valeur de quantum pour chaque type d'application. Ensuite, en temps réel, il associe un vCPU en cours d'exécution à un type d'application. Enfin, regroupe les vCPUs en fonction de leur affinité de quantum (valeur de quantum obtenue durant le calibrage) et uti-

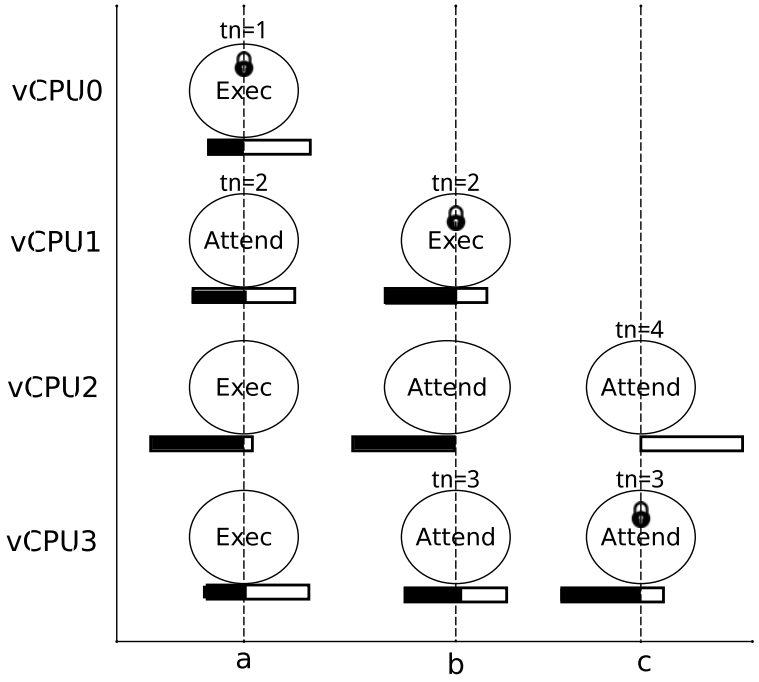

FIGURE 3.8 – Illustration du fonctionnement de I-Spinlock.

lise cette valeur de quantum pour ordonnancer les vCPUs. Une évaluation de *AQL_Sched* montre qu'il peut améliorer les performances des applications jusqu'à 20% comparé à l'hyperviseur Xen.

3.4 Ordonnanceur collaboratif : I-Spinlock

Dans la première contribution que nous venons de présenter, l'idée consiste à résoudre le problème de préemption de Spinlock et gestion des interruptions au niveau de l'hyperviseur. Dans cette section, nous proposons une autre contribution qui préconise d'agir plutôt au niveau de l'OS invité. Cette contribution aborde uniquement les problèmes liés au Spinlock (LHP et LWP) sans la gestion des interruptions. Nous proposons pour résoudre ces problèmes une nouvelle primitive de Spinlock, Informed-Spinlock (appelé I-Spinlock) dans l'OS invité.

3.4.1 Informed Spinlocks (I-Spinlock)

Nous débutons par une description de l'idée de base derrière I-Spinlock. Ensuite, nous détaillons la conception de I-Spinlock.

3.4.1.1 Description générale

Avant la présentation de notre idée, rappelons les deux problèmes que nous abordons dans cette section. Le problème de LHP se produit lorsque le vCPU d'un thread tenant un verrou est préempté par l'ordonnanceur de l'hyperviseur. Le problème de LWP se produit lorsque le vCPU d'un thread ayant un ticket est préempté et tous ses prédécesseurs ont déjà obtenu le verrou. Par conséquent, nous pouvons dire qu'un problème de LHP ou LWP survient lorsque le vCPU d'un thread contenant soit un verrou soit un ticket est préempté. De ce fait, les problèmes de LHP et LWP sont philosophiquement similaires. **Leur origine est le fait qu'un thread est autorisé à acquérir un verrou/ticket tandis que le reste du quantum de son vCPU n'est pas suffisant pour acquérir et compléter la section critique.** Par conséquent, l'idée que nous développons, consiste à éviter cette situation. **Pour ce faire, nous proposons de ne permettre à un thread d'acquérir un ticket si et seulement si le quantum restant de son vCPU est suffisant pour entrer et quitter la section critique.** Notons que cette contrainte sur l'acquisition du ticket est suffisante, car un thread prend un ticket avant d'acquérir un verrou.

Pour atteindre son objectif, I-Spinlock introduit une nouvelle métrique (associée à chaque vCPU) appelée *capacité de complétion* (*lock_cap* pour abréger) qui indique la capacité de son thread (thread s'exécutant dans le vCPU) à prendre un ticket, attendre son tour, acquérir le verrou, exécuter la section critique et relâcher le verrou avant la fin du quantum. Soit un vCPU v, son *lock_cap* est calculé comme suit :

$$lock\_cap(v) = r\_ts(v) - (lql + 1) \times csd \tag{3.6}$$

où $r\_ts(v)$ est le quantum restant à v avant sa préemption, lql est le nombre total de threads en attente du verrou (ils ont déjà des tickets) et csd est la durée de la section critique. A partir de cette formule, I-Spinlock fonctionne comme suit. Chaque fois qu'un thread tente de prendre un ticket, l'OS invité calcule le *lock_cap* de ce vCPU. Si la valeur obtenue est supérieure à zéro (le vCPU a suffisamment de temps avant sa préemption), le thread est autorisé à prendre un ticket. Sinon, il n'est pas autorisé et il doit attendre le prochain quantum de son vCPU. Ces threads sont appelés *threads incapables* dans I-Spinlock. Notre approche est similaire à celle présentée dans [53] pour les systèmes non virtualisés, appelée "two-minute warning" [3] La figure 3.8 illustre le fonctionnement de I-Spinlock par étapes avec un exemple simple. Pour ce faire, nous considérons une VM configurée avec 4 vCPUs (vCPU0-vCPU3). Pour illustrer comment fonctionne I-Spinlock, supposons que le *csd* représente le quart du quantum. Initialement (étape a), le verrou est libre. Les threads vCPU0 et vCPU1 tentent de prendre un ticket. Le

3. [53] fournit un avertissement peu avant la préemption. I-Spinlock suit la même direction en appliquant cette approche aux environnements virtualisés.

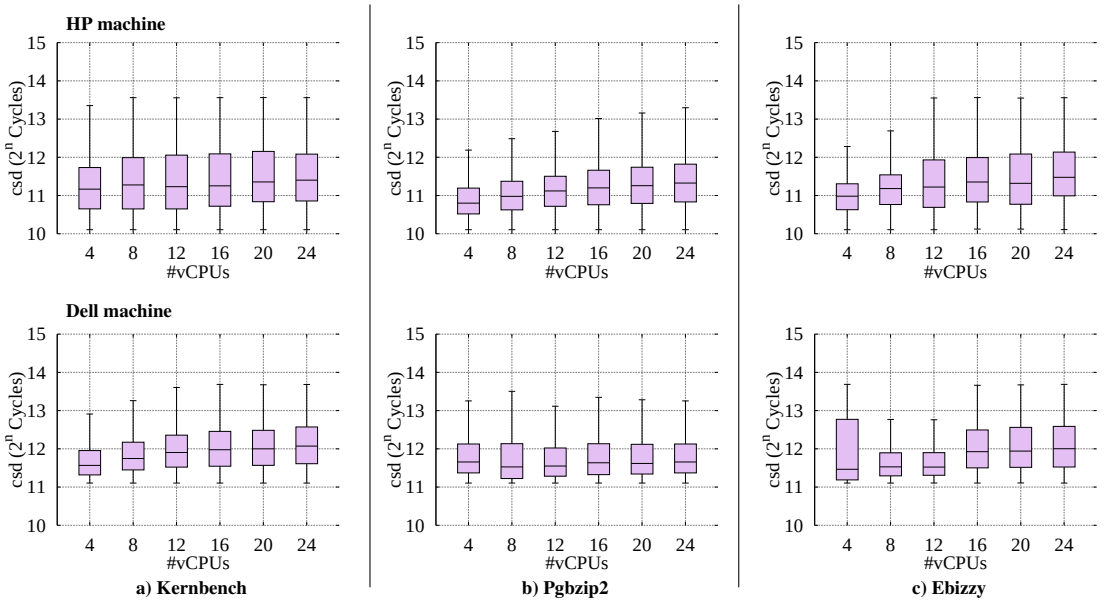

FIGURE 3.9 – Estimation de la durée de la section critique (csd).

thread du vCPU0 est le premier à prendre le ticket, donc le verrou. Le thread du vCPU1 prend le deuxième numéro de ticket, car son *lock_cap* indique que son quantum restant suffit pour terminer la section critique avant d'être préempté. Il entre dans une phase d'attente active. Le thread de vCPU2, suivi du thread de vCPU3, tente de prendre un ticket. Puisque $lock\_cap(vCPU2)$ montre qu'il n'est pas capable de terminer la section critique dans son intervalle de temps résiduel, son thread n'est pas autorisé à prendre un ticket et il est marqué *incapable* (section 3.4.1.2 détaille les actions prises par les threads *incapable*). Ce n'est pas le cas pour le thread du vCPU3 qui prend le numéro de ticket 3 (étape b). Au stade c, vCPU2 démarre un nouveau quantum, résultant en une valeur positive pour son $lock\_cap$. Le thread de vCPU2 peut prendre un ticket.

Nous constatons, théoriquement, qu'un OS qui utilise I-Spinlock ne souffre ni de LHP ni de LWP. Cependant, l'implémentation de cette solution soulève deux principaux défis qui sont résumés par les questions suivantes :

— $r\_ts(v)$: comment l'OS invité peut-il être informé du quantum restant de ses vCPUs, sachant que cette information n'est disponible qu'au niveau de l'hyperviseur ?

— *csd* : comment l'OS invité peut-il estimer le temps nécessaire pour compléter une section critique, sachant que sa durée n'est pas toujours la même ?

Disponibilité du quantum restant pour un vCPU($r\_ts(v)$) dans l'OS invité
Nous ne considérons plus l'hyperviseur comme une boîte noire comme c'est tra-

ditionnellement le cas. I-Spinlock implémente un mécanisme de mémoire partagée entre l'hyperviseur et l'OS invité, ce qui conduit à un hyperviseur qui est une boîte grise. De cette façon, l'hyperviseur peut partager des informations avec les OS invités. Dans notre cas, cette information est limitée à la fin du quantum d'un vCPU. Ceci n'est pas critique pour la sécurité de l'hyperviseur, ni des VMs. Par conséquent, chaque fois qu'un vCPU est ordonnancé, l'ordonnanceur de l'hyperviseur met dans la mémoire partagée le temps exact de la future préemption du vCPU. C'est ce qu'on appelle le *temps de préemption*. Ce faisant, $r\_ts(v)$ peut être calculé dans l'OS invité chaque fois qu'un thread veut prendre un ticket. Le calcul de $r\_ts(v)$ est donné par la formule suivante :

$$r\_ts(v) = preemption\_time - actual\_time; \tag{3.7}$$

où $actual\_time$ est le temps courant (lorsque le thread prend le ticket).

Calcul de la durée de la section critique (*csd*)
Comme plusieurs travaux de recherche précédents [41, 80], nous estimons *csd* par calibrage. À cette fin, nous réalisons un ensemble d'expérimentation avec différentes applications (Kernbench, Ebissy et Pgbzip2, présentés dans la section 3.4.2) avec différentes valeurs de vCPUs. Les expériences ont été menées sur différents types de machines modernes et nous observons les mêmes résultats. La figure 3.9 présente les résultats obtenus pour deux types de machines : DELL et HP (leurs caractéristiques sont présentées dans la section 3.4.2). Tout d'abord, nous reconnaissons que la moyenne du *csd* est d'environ 2^{12} cycles. Cela correspond à ce qui a été rapporté par d'autres études, réalisé sur d'autres types de machines [41, 80]. Afin de couvrir toutes les durées de section critiques, I-Spinlock utilise 2^{15} cycles comme valeur de *csd* (correspondant à 6 microsecondes sur nos machines). Cette valeur est supérieure à la valeur maximale mesurée de *csd* (figure 3.9). Ce faisant, nous minimisons le nombre de faux négatifs entre les threads *incapables*.

3.4.1.2 L'implémentation de I-Spinlock

Bien que notre solution puisse être appliquée à tous les systèmes de virtualisation, cette section présente la mise en œuvre d'un prototype dans Xen 4.2.0 et le noyau Linux version 3.13.0. Cette implémentation implique à la fois l'hyperviseur et l'OS invité. Notre ambition est de fournir une implémentation qui fonctionne autant dans les VMs de type HVM que PV. L'implémentation HVM s'appelle HVM I-Spinlock alors que l'implémentation PV s'appelle PV I-Spinlock.

L'implémentation de la mémoire partagée
Notre implémentation profite du mécanisme de mémoire partagée qui existe déjà

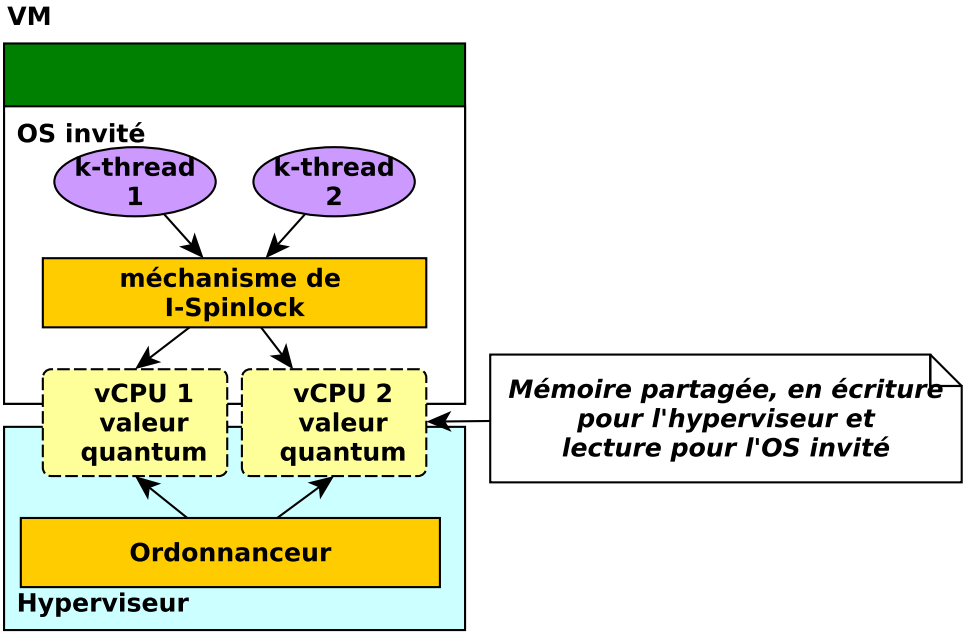

FIGURE 3.10 – Disponibilité du quantum restant ($r\_ts(v)$) dans l'OS invité pour I-Spinlock.

dans Xen (ainsi que d'autres systèmes de virtualisation). I-Spinlock exploite spécialement la structure de données *shared_info*.

Structure de données shared_info de Xen. Xen utilise une page de mémoire partagée (appelée *shared_info*) pour fournir aux VMs des informations matérielles telles que la taille de la mémoire, nécessaire au processus de démarrage du noyau. Chaque VM a son propre *shared_info* qui est utilisé une fois par l'hyperviseur (au moment de la création de la VM) et l'OS (au moment du démarrage du noyau). La façon dont le mappage de *shared_info* est établi diffère entre PV et HVM. Dans PV, *shared_info* est attribué par l'hyperviseur et apparaît à une adresse virtuelle fixe dans l'espace d'adressage de l'OS invité. L'adresse machine correspondante est communiquée à l'OS invité via la structure de données *xen_start_info*. En mode HVM, l'OS invité a un contrôle total sur son espace d'adressage physique. Il peut allouer la structure de données *share_info* dans l'une de ses pages physiques. L'adresse physique choisie est communiquée de l'OS invité à l'hyperviseur.
Utilisation de shared_info dans I-Spinlock. L'ordonnanceur de l'hyperviseur est modifié de sorte que chaque fois qu'un vCPU est mis sur un pCPU, le temps de fin du quantum est stocké dans la structure de données *shared_info* de sa VM. Le temps de fin du quantum est fourni en cycles de CPU. Chaque vCPU possède une entrée dédiée dans *shared_info*.

3.4.1.3 L'acquisition du ticket et du verrou

Notre implémentation intervient comme une amélioration des tickets Spinlocks de Linux. L'algorithme de la figure 3.11 présente le patch qui doit être appliqué à l'OS invité et se comprend comme suit. La routine *arch_spin_lock* est invoquée chaque fois qu'un thread tente d'accéder à un Spinlock. Le temps de fin du quantum d'un vCPU, stocké dans la page partagée par l'hyperviseur, est lu à la ligne 16. À l'aide de cette information, le temps restant ($r\_ts(v)$) est calculé à la ligne 18 selon l'équation 3.7. Le nombre de vCPU en attente du verrou (lql) est calculé à la ligne 20. lql est obtenu en soustrayant le *head* du *queue* du ticket. Le *lock_cap* du vCPU est calculé à la ligne 22. S'il est négatif, le thread est marqué comme *incapable*. Un thread *incapable* peut prendre deux chemins possibles en fonction du mode de virtualisation. Il effectue une attente active dans le cas des VMs de type HVM (lignes 29-33) ou il libère le processeur dans le cas des VMs de type PV (ligne 35). La libération du processeur est possible en mode PV, car il permet l'utilisation les hypercalls. Par conséquent, au lieu d'entrer dans une phase d'attente active (mode HVM), le thread *incapable* invoque un hypercall (mode PV) qui indique à l'hyperviseur de retirer le vCPU du pCPU. En ce sens, PV I-Spinlock évite le gaspillage des cycles du CPU à la différence de HVM I-Spinlock. Les résultats de l'évaluation montrent que cette optimisation rend PV I-Spinlock plus efficace que HVM I-Spinlock.

3.4.2 L'évaluation de I-Spinlock

Cette section présente les résultats d'évaluation de I-Spinlock. Nous évaluons les aspects suivants.

— *Risques de famine et d'injustice.* La capacité d'I-Spinlock à éviter à la fois la famine et l'iniquité.

— *Efficacité.* La capacité de I-Spinlock à résoudre les problèmes LHP et LWP.

— *Générique.* La capacité d'I-Spinlock à tenir compte des types des VM PV et HVM.

— *Overhead.* La quantité de ressources consommées par I-Spinlock.

— *Positionnement.* La comparaison de I-Spinlock avec d'autres solutions.

Nous présentons d'abord l'environnement expérimental avant la présentation des résultats de l'évaluation.

```
1 #define AVERAGE_HOLD_TIME (1 << 15)
2 /*Get the actual time*/
3 +uint64_t rdtsc(void)
4 +{
5 +   unsigned int hi, lo;
6 +   __asm__volatile("rdtsc" : "=a" (lo), "=d" (hi));
7 +   return ((uint64_t)hi << 32) | lo;
8 +}
9
10 void arch_spin_lock(arch_spinlock_t *lock)
11 {
12    struct __raw_tickets inc = {.tail = TICKET_LOCK_INC};
13     /*Share_info page*/
14 +  struct shared_info *s = HYPERVISOR_shared_info;
15     /*Compute the remain time_slice*/
16 +  uint64_t time=rdtsc();
17 +  uint32_t cpu= cpuid();
18 +  uint64_t remaining_slice= s->vcpu_info[cpu].time_slice-time;
19         /*Compute the number of threads waiting for the lock*/
20 +  uint8_t dist = lock->tickets.head - lock->tickets.tail;
21     /*Compute the vCPU's lock_cap*/
22 +  int64_t lock_cap= remaining - (dist + 1) * AVERAGE_HOLD_TIME ;
23     /*Decide if the thread is allowed to take a ticket*/
24 +  if(lock_cap < 0)
25 +  {
26         /*The thread is not allowed. It should wait the next quantum to take a ticket*/
27 +   #ifdef CONFIG_HVM_SPINLOCKS
28
29 +    u64 slice_ID = s->vcpu_info[cpu].slice_ID;
30 +    do
31 +    {
32 +           cpu_relax();
33 +        }while(slice_ID==s->vcpu_info[cpu].slice_ID);
34 +    #else
35 +    __halt_cpu(cpu);
36 +    #endif
37
38 +  }
39    inc = xadd(&lock->tickets, inc);
40    if(likely(inc.head == inc.tail))
41      goto out;
42    for (;;)
43    {
44      do
45      {
46        if (ACCESS_ONCE(lock->tickets.head)==inc.tail)
47                goto out;
48        cpu_relax();
49      }
50    out:
51    barrier();
52 }
```

FIGURE 3.11 – Implémentation de I-Spinlock dans l'OS invité.

Procédure expérimentale et configurations	
Nous exécutons deux instances du meme benchmark dans deux VMs, les deux sont soit en HVM, soit en PV. Nous utilisons les VMs type HVM durant l'évaluation de HVM IS et les VMs PV durant l'évaluation de PV IS. Tous les vCPUs partagent le même pool de pCPUs durant les évaluations.	
Abréviations	
TSL	Ticket Spinlock
HVM IS	HVM I-Spinlock
PV IS	PV I-Spinlock
PTS	Preemptable Ticket Spinlock
TS	Time Slice
PV TS	Para virtual Ticket Spinlock
OT	Oticket

TABLE 3.6 – Configurations expérimentales : procédure expérimentale et configurations, et abréviations adoptées.

3.4.2.1 L'environnement expérimental

Pour démontrer l'efficacité de I-Spinlock, il est évalué avec différents types de benchmark, fonctionnant sur différents types de machines. Nous utilisons à la fois des micro-benchmarks et des macro-benchmarks qui ont été utilisés dans beaucoup d'autres travaux [63].

Benchmarks

— Un micro-benchmark, développé pour les besoins de ce travail. Il lance n threads qui tentent d'accéder x fois à la même zone de mémoire qui est la session critique. Le nombre de threads est le nombre de vCPU de la VM. La performance de l'application est donnée par deux paramètres, à savoir le temps d'exécution moyen de tous les threads et l'écart-type. Ce benchmark est conçu pour effectuer beaucoup d'accès en parallèle, afin de produire beaucoup de contention.

— Kernbench [10] est un benchmark CPU intensif qui consiste en une compilation parallèle du noyau Linux. Sa performance est donnée par la durée de la compilation (exprimée en secondes).

— Pbzip2 [13] est une implémentation parallèle de bzip2, le compresseur de fichiers qui est présent dans les systèmes Linux. Il utilise pthreads et réalise une compression parallèle sur les machines SMP. Sa performance est donnée par la durée de la compression (en secondes).

— Ebizzy [4] génère une charge de travail ressemblant aux charges de travail communes des serveurs d'applications Web. Il exécute plusieurs threads,

dispose d'un grand WSS, et alloue et désactive la mémoire fréquemment. Sa performance est mesurée avec un débit (records/seconde).

Le matériel

Les expériences ont été effectuées sur deux types de machines exécutant la même distribution Linux (serveur ubuntu 12.04) :

— Dell : 24 cœurs (hyper thread), 2.20 GHz Intel Xeon E5-2420, 16 Go de RAM, 15 MB LLC.

— HP : 8 cœurs, 3,2 GHz Intel Core i 7, 16 Go de RAM, 8 MB LLC

Les résultats obtenus à partir des deux machines sont presque les mêmes. Par conséquent, cette section ne présente que les résultats de la machine DELL.

La procédure expérimentale

Sauf indication contraire, chaque expérience se réalise comme suit. Nous exécutons deux instances du même benchmark dans deux VMs. L'exécution est ensuite répétée avec différents nombres de vCPUs. Tous les vCPUs partagent le même groupe de pCPUs. Toutes les VMs utilisent un système de fichiers en mémoire (*tmpfs*) pour atténuer l'effet secondaire des opérations d'E/S. Toutes les expériences ont également été répétées 5 fois. Le tableau 3.6 résume les abréviations utilisées dans cette section.

3.4.2.2 Risques liés à la famine et à l'injustice

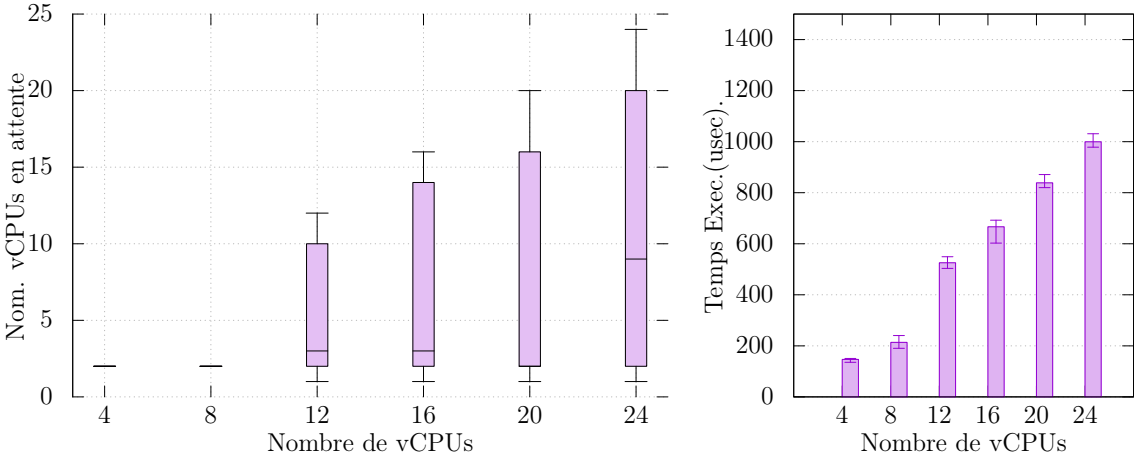

FIGURE 3.12 – Évaluation du risque de famine et iniquité dans I-Spinlock.

Ticket Spinlock utilise la politique du premier arrivé, premier servi (FCFS) afin d'éviter la famine d'une part et d'assurer l'équité d'autre part. Sachant que I-Spinlock altère la politique FCFS, on peut légitimement demander comment I-Spinlock traite de la famine et de l'équité. Cette section traite et évalue ces aspects

(nous verrons dans la section 3.4.2.5 qu'il s'agit d'un problème pour d'autres solutions).

La politique FCFS est mise en place dans les Ticket Spinlocks en imposant à chaque thread de prendre un ticket avant d'essayer d'obtenir un verrou. Ceci n'est pas complètement altéré dans I-Spinlock, car il utilise toujours les tickets sauf pour les threads qui sont marqués *incapable*. Par conséquent, seuls les threads *incapables* sont concernés par les risques de famine et d'injustice. Nous expliquons dans le reste de ce paragraphe que cette situation n'est pas problématique. À cette fin, nous montrons que dans I-Spinlock, un thread *incapable* n'attendra pas longtemps avant de prendre un ticket. En effet, la combinaison de deux facteurs permet à I-Spinlock d'éviter naturellement la famine et l'inéquité. Tout d'abord, lorsqu'un vCPU *incapable* reprend le processeur, il lui est attribué un nouveau quantum et le thread *incapable* sera exécuté en premier. C'est parce que l'ordonnanceur de l'OS invité ne peut pas préempter le thread *incapable* puisqu'il est dans la routine Spinlock. Il s'agit du fonctionnement traditionnel de l'ordonnanceur du noyau envers les threads qui utilisent des Spinlocks. Le deuxième facteur est le fait que le quantum d'un vCPU est bien supérieur à la durée de la section critique (*csd*), ce qui entraîne une valeur positive de *lock_cap* au début de tout quantum. En effet, la valeur du quantum est de l'ordre de la milliseconde (par exemple, 30ms dans Xen et 50ms dans VMware) alors que *csd* est de l'ordre de la microseconde (2^{15} Cycles qui correspondent à 6 microsecondes sur notre machine), voir la figure 3.9 (dans la section 3.3.1.1). Par conséquent, la probabilité d'avoir un *lock_cap* négatif au début du quantum d'un vCPU est pratiquement nulle. Ce pourrait être autrement dans le cas d'une VM avec $\frac{quantum\ length \times 1000}{6}$ vCPUs ayant un ticket (correspondant à la valeur de *lql* dans l'équation 3.6). Cela représente environ 5000 vCPU actifs dans un OS invité Xen et environ 8333 vCPUs dans un OS invité VMware, ce qui est pratiquement impossible (c'est plutôt de l'ordre d'une centaine de vCPUs). Pour valider cette explication, nous utilisons le micro-benchmark qui génère beaucoup de contention pour l'accès à un objet. Nous nous intéressons à trois mesures, à savoir : la valeur de *lql*, le temps d'exécution moyen de tous les threads et l'écart type. La figure 3.12 présente les résultats. Nous faisons les constats suivants. Tout d'abord, en raison du nombre de verrous, *lql* peut être aussi important que le nombre de vCPU (figure 3.12 de gauche). Deuxièmement, dans chaque évaluation, pour un nombre de vCPU donné, tous les threads s'exécutent à peu près pour la même durée (la figure 3.12 de droite), puisque l'écart-type est presque nul (le petite barre sur les histogramme). Cela signifie que les threads acquièrent le verrou avec une latence raisonnable. En résumé, I-Spinlock garantit qu'un thread précédemment *incapable* obtiendra toujours le ticket lors du prochain quantum de son vCPU, évitant ainsi la famine et l'iniquité.

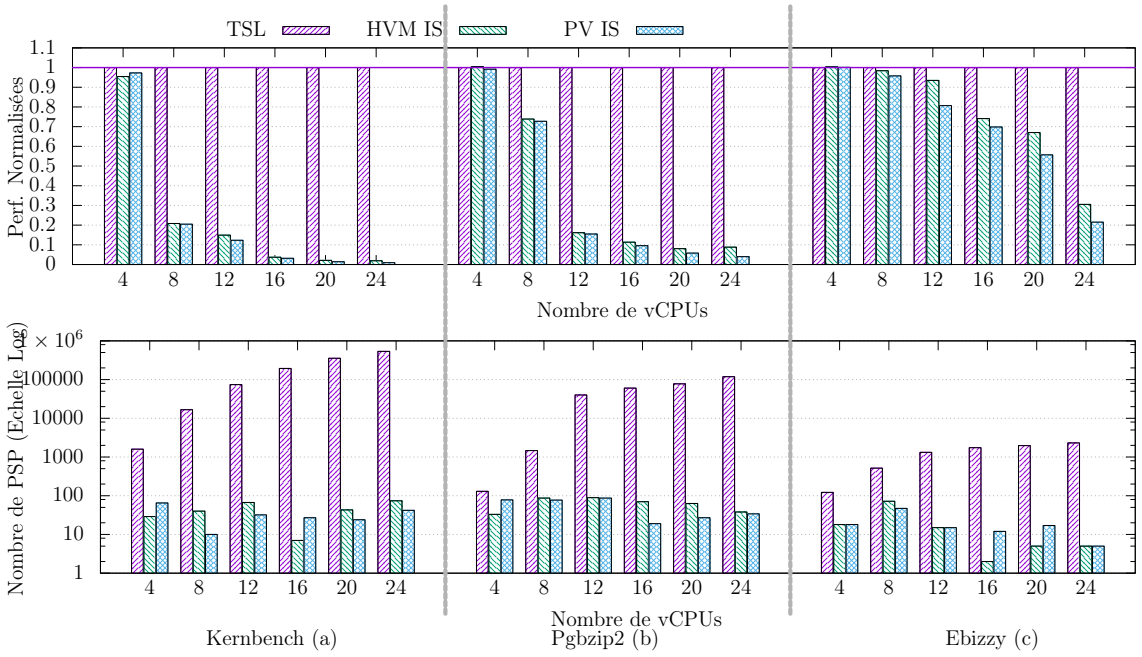

FIGURE 3.13 – Capacité de I-Spinlock à résoudre les problèmes de LHP et LWP.

3.4.2.3 L'efficacité de I-Spinlock

La métrique qui permet d'évaluer l'efficacité d'I-Spinlock est le nombre de *préemption problématique de Spinlock* (PSP), qui est le nombre de LHP et de LWP. Nous collectons la valeur de cette métrique dans trois situations : le Ticket Spinlock standard (la référence notée TSL), HVM I-Spinlock (noté HVM IS) et PV I-Spinlock (PV IS). La figure 3.13 présente les résultats de ces expériences. La courbe supérieure présente les performances normalisées (sur la référence) des benchmarks étudiés tandis que la courbe inférieure présente le nombre total de PSP. On peut voir sur la courbe inférieure que le nombre de PSP (LHP + LWP) augmente avec le nombre de vCPU pour le ticket Spinlock. I-Spinlock ne souffre pas de ces problèmes (le nombre de PSP est beaucoup plus petit). On remarque également que le nombre de PSP avec I-Spinlock est maintenu constant lorsque le nombre de vCPU augmente, ce qui illustre le passage à l'échelle de I-Spinlock. PV IS fonctionne un peu mieux que HVM IS, car les threads *incapables* perdent des cycles CPU en mode HVM pendant qu'ils relâchent le processeur en mode PV (voir la section 3.4.1.3). Enfin, notons que I-Spinlock ne peut pas totalement empêcher les PSP, car plusieurs threads peuvent calculer leur $lock\_cap$ en même temps, leurs valeurs de lql peuvent être erronées et csd n'est qu'une moyenne.

3.4.2.4 Overhead de I-Spinlock

Nous évaluons l'overhead de HVM IS et PV IS en terme de nombre de cycles CPU dont chacun a besoin. Pour ce faire, nous exécutons kernbench dans une VM 24 vCPUs. Nous utilisons une seule VM afin d'éviter les problèmes de LHP et LWP. La figure 3.14 de droite montre le nombre de cycles CPU nécessaires à l'acquisition d'un verrou tandis que la figure 3.14 de gauche présente le temps d'exécution normalisé du benchmark. La normalisation est réalisée sur les résultats du Ticket Spinlock. Nous pouvons constater que le nombre de cycles CPU utilisé par HVM IS ou PV IS est légèrement supérieur au nombre de cycles CPU utilisés par le Ticket Spinlock. Cela provient des opérations supplémentaires introduites par I-Spinlock. À partir de la figure 3.14 de gauche, nous pouvons voir que l'impact de cet overhead est négligeable car I-Spinlock et le ticket Spinlock conduisent aux mêmes résultats. En résumé, l'overhead de notre solution est presque nul.

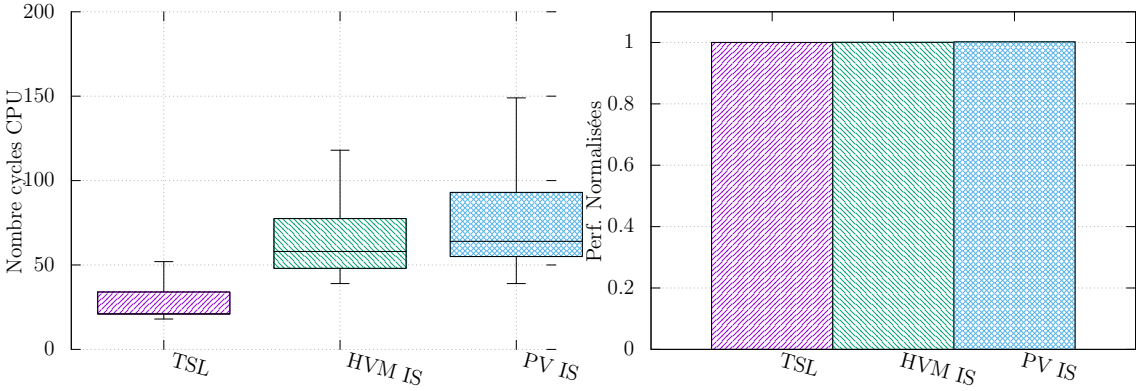

FIGURE 3.14 – Estimation de l'overhead de I-Spinlock.

3.4.2.5 La comparaison avec d'autre solutions

Nous comparons I-Spinlock avec des solutions existantes présentées dans l'état de l'art.

Solutions compatibles avec la virtualisation assistée par le matériel
Nous comparons HVM IS avec les deux solutions compatibles avec des VMs de type HVM existantes, à savoir les preemptable tickets (notés PTS) [63] et le microsliced [23] qui est la réduction du quantum (noté TS). La figure 3.15 présente les performances normalisées de kernbench, normalisées sur HVM IS. Commençons par analyser les résultats de PTS. PTS fournit presque les mêmes performances que HVM IS avec un petit nombre de vCPU pour la VM (4 et 8 vCPUs). Cependant, il ne s'améliore pas avec les VMs plus grandes (nombre de vCPUs

> 12). Cela s'explique par le fait que PTS permet une acquisition désordonnée du verrou lorsque les threads précédents (les premiers threads avec des tickets) sont préemptés. Par conséquent, plus le nombre de vCPU croit (augmentation de la contention), plus il y a des threads essayant d'acquérir simultanément le même verrou sans ordre établi. Cela ramène PTS aux anciennes implémentations de Spinlock (voir la section 3.1.1) où le thread le plus rapide va toujours acquérir le verrou, ce qui conduit à des situations de famine. En outre, PTS ne traite pas le problème de LHP. En ce qui concerne microsliced (TS) [23], nous constatons qu'il surpasse légèrement HVM IS lorsque le nombre de vCPU est faible (nombre de vCPUs < 16). Nous pouvons constater que le nombre de PSP n'a pas d'effet significatif sur TS. En effet, avec un petit quantum, le temps gaspillé par le vCPU en raison de PSP est réduit. Cependant, comme nous l'avons dit au début du chapitre, réduire le quantum augmente le nombre de commutations de contexte, qui est connu pour être négatif pour les VMs utilisant énormément de CPU [86].

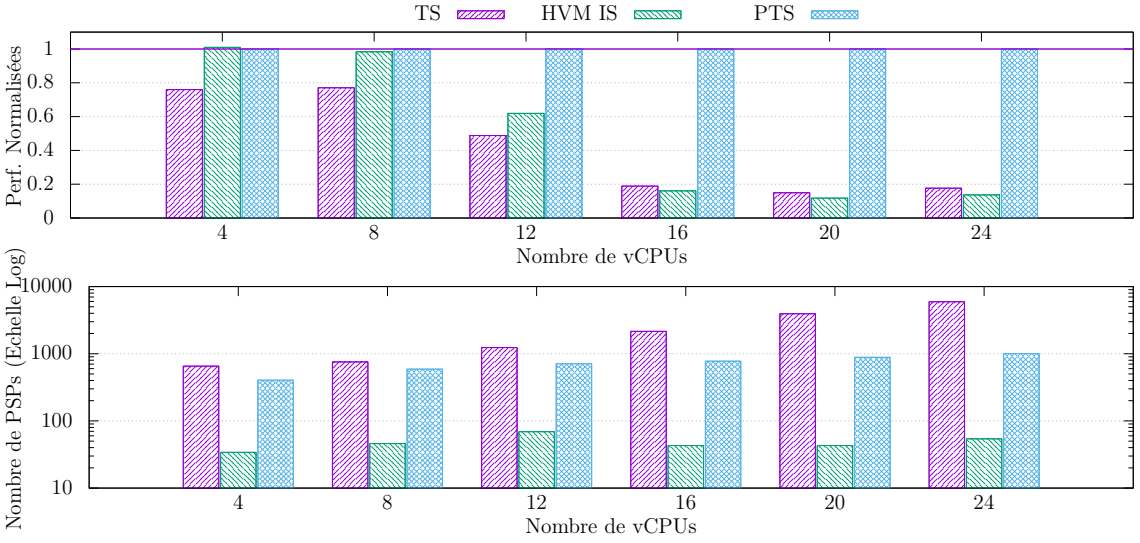

FIGURE 3.15 – Comparaison entre HVM I-Spinlock (HVM IS) et les solutions compatibles avec la virtualisation assistée par le matériel.

Solutions compatibles avec la para virtualisation Nous comparons également PV IS aux deux principales solutions compatibles PV existantes, à savoir le para-virtualized ticket [12] (noté PV TS) et Oticket [49] (noté OT). La figure 3.16 de haut présente les performances normalisées de kernbench, normalisées sur PV TS. Nous pouvons constater que toutes les solutions fournissent sensiblement les mêmes résultats avec des scénarios de contention faible (faible nombre de vCPU). Ce n'est pas le cas avec des scénarios de contentions lourdes où seule notre solution offre de meilleurs résultats. Cela s'explique par le fait que le nombre de PSP

augmente linéairement avec le nombre de vCPU dans d'autres solutions alors qu'il est presque constant dans PV IS.

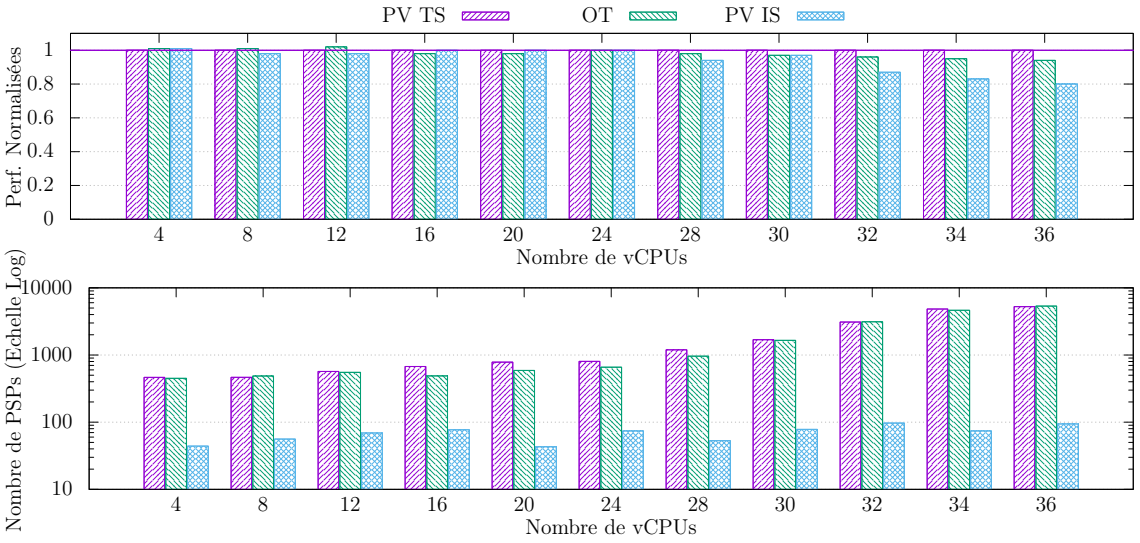

FIGURE 3.16 – Comparaison de PV I-Spinlock (PV IS) avec les solutions compatibles avec la para virtualisation.

3.4.3 Synthèse

Nous venons de présenté I-Spinlock, une nouvelle implémentation des tickets spinlock ciblant les systèmes virtualisés. Le principe de I-Spinlock est de ne permettre à un thread d'acquérir un ticket que si le reste du quantum de son vCPU est suffisant pour attendre son tour (selon son numéro de ticket), entrer et quitter la section critique. Cette précaution permet d'éviter tout LHP ou LWP. Afin de démontrer efficacité de I-Spinlock, nous l'avons évalué à l'aide de plusieurs benchmarks (Kerbench, Pbzip2, Ebissy). Nous avons comparé I-Spinlock avec d'autres solutions existantes et les résultats obtenus attestent que I-Spinlock résout le problème de LHP et LWP.

3.5 Synthèse

Dans ce chapitre, nous avons présenté deux défis importants de l'ordonnancement dans les systèmes virtualisés. Le partage du CPU entre les différentes VMs provoque une discontinuité durant l'exécution des vCPUs sur les pCPUs. Cette

discontinuité entraîne une inefficacité significative pour les mécanismes très importants de l'OS tels que les spinlock et des interruptions. En effet, ces mécanismes reposent sur la disponibilité immédiate des CPUs chaque fois que l'OS a besoin ce qui n'est pas le cas dans un système virtualisé. Pour répondre à ces problèmes dus à cette discontinuité, nous avons présenté deux contributions. La première contribution propose un nouvel ordonnanceur de l'hyperviseur qui agit sur la valeur du quantum comme levier de solution aux problèmes. Cet ordonnanceur s'appelle *AQl_Sched*. La seconde contribution préconise une collaboration entre l'ordonnanceur de l'hyperviseur et l'OS invité. L'ordonnanceur de hyperviseur peut en effet transmettre aux VMs certaines informations d'ordonnancement, pour que ces dernières puissent agir efficacement et éviter ces problèmes. Une instanciation de cette approche est une nouvelle primitive de Spinlock que nous avons appelée I-Spinlock.

Chapitre 4

Imprévisibilité des performances dans un Cloud virtualisé

Contents

Nous présentons dans ce chapitre, le problème d'imprévisibilité des performances dans un Cloud virtualisé. Nous détaillons les différentes causes de ce problème d'imprévisibilité des performances. Par la suite, nous décrivons deux contributions pour résoudre ce problème. Les descriptions des contributions sont suivies par des évaluations. Nous concluons le chapitre par une synthèse.

4.1 Motivation

Dans un Cloud fournissant un service du type IaaS, un client peut faire la réservation et l'utilisation à la demande d'une quantité de ressources sur une infrastructure matérielle distante. Afin d'isoler, d'allouer et de garantir des quantités de ressources dans un IaaS, les fournisseurs de Cloud utilisent la virtualisation, et les VMs sont les unités d'allocations. Le fournisseur établit un catalogue des types de VMs (exemple t2.medium, t2.small chez Amazon EC2) présentant les différentes configurations de VMs pouvant être réservées par les clients. La configuration d'une VM définit les quantités de ressources, qui sont allouées à la VM vis-à-vis des différents périphériques : CPU, mémoire, réseau et disque. En général, les ressources d'une VM sont spécifiées comme suit.

— capacité réseau en débit (exemple 1000 Mega octet/s),
— capacité disque en débit d'E/S (exemple 100 Mega octet/s) et d'espace de stockage (exemple 1 Tera octet),
— capacité mémoire en quantité d'espace mémoire (exemple 10 Giga octets),
— enfin, capacité CPU en nombre de vCPU.

Ces ressources allouées à la VM correspondent à une qualité de service, mentionnées dans le SLA, liant le fournisseur et le client. Ce dernier s'attend à un respect du SLA à tout instant. L'imprévisibilité des performances est la conséquence immédiate de l'incapacité du fournisseur à garantir le SLA. Nous parlons du problème d'imprévisibilité des performances si les performances d'une application s'exécutant dans une VM du Iaas varient dans le temps. En d'autres termes, le fournisseur ne garantit pas l'isolation des performances. Le problème d'imprévisibilité des performances est identifié par Microsoft [58] comme faisant partie des 5 problèmes majeurs dans le Cloud. Deux principales causes en sont à l'origine :

— un mauvais partage des ressources entre les différentes VMs,
— l'hétérogénéité des infrastructures dans les centres d'hébergement.

4.1.1 Les ressources partagées comme source d'imprévisibilité des performances

Un système virtualisé dispose de plusieurs ressources partagées, des ressources matérielles telles que le CPU, le disque, la mémoire et la carte réseau, mais également des ressources logicielles telles que les VMs contenant les pilotes des périphériques, que nous appellerons DD (pour Domain Driver). L'hyperviseur effectue le partage de toutes ces ressources entre toutes les VMs, mais doit également garantir l'isolation des performances. En d'autres termes, l'activité d'une VM ne doit pas influencer l'activité d'autres VMs. Les ressources telles que le CPU, la mémoire et le disque disposent de mécanismes efficaces pour leur partage et l'isolation des performances (l'ordonnanceur de l'hyperviseur pour le CPU, la MMU pour la mémoire). Mais ceci n'est pas le cas pour certaines ressources tels que le DD ou encore les caches du CPU qui sont des ressources micro-architecturales. En effet, en ce qui concerne les caches du CPU, les hyperviseurs actuels (Xen, VMware ou kvm) ne disposent pas de solution de partage entre les différentes VMs. Ainsi, les VMs doivent concourir pour l'accès aux caches sans règle de partage, ce qui conduit au problème d'imprévisibilité des performances ; l'activité d'une VM pourait influencer les autres. Ceci explique pourquoi nombreux sont les travaux [78, 39, 69] qui se sont intéressés au partage du cache matériel entre les VMs comme principale source du problème d'imprévisibilité des performances. En revanche, très peu de travaux se sont penchés sur les ressources logicielles partagées comme origine du problème d'imprévisibilité des performances. Pour mieux comprendre comment le partage des ressources logicielles peut y conduire, il est important de faire quelques rappels sur la gestion des périphériques d'E/S dans les hyperviseurs, notamment dans Xen.

Les E/S dans Xen et le modèle de Split-Driver

Dans le modèle para-virtualisé de Xen, le pilote de chaque périphérique d'E/S réside dans une VM particulière qu'on appelle DD. Le DD traite les opérations d'E/S pour le compte des VMs. Les VMs exécutent un faux pilote appelé frontend *(FE)*, comme le montre la figure 4.1. Le frontend communique avec le pilote réel par l'intermédiaire d'un Backend (BE) (module dans le DD) qui permet le partage de la ressource matérielle. Cette architecture de virtualisation des E/S est utilisée par la majorité des systèmes de virtualisation et est connue comme le «*modèle split-driver*». Ce modèle est présenté dans la figure 4.1 et fonctionne comme suit. Le pilote réel dans le DD peut accéder au matériel, mais les interruptions sont traitées par l'hyperviseur. Les communications entre le DD et l'hyperviseur, et entre le DD et un OS invité sont fondées sur les canaux d'événements (event channels en anglais) (EC1 et EC2 dans la figure 4.1). Une requête d'E/S émise par un OS invité crée un événement sur EC1. Les données sont transmises au

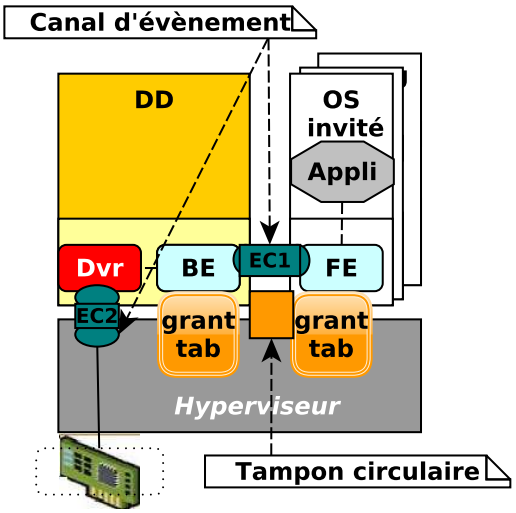

FIGURE 4.1 – Le modèle de Split-Driver pour la virtualisation des E/S

BE dans le DD à travers un tampon circulaire (fondé sur un partage de pages mémoire entre l'OS invité et le DD). Le BE est alors responsable d'invoquer le pilote réel. Une requête d'E/S reçue par le matériel génère un événement sur EC2. À partir de cet événement, l'hyperviseur génère une interruption virtuelle qui est gérée par le vrai pilote. L'OS dans le DD est configuré pour que le BE soit la destination de transfert pour toute demande d'E/S.

Problèmes et conséquences

Selon le modèle du split-driver, nous constatons que les requêtes d'E/S sont traitées par l'hyperviseur et le DD pour le compte des VMs. Par conséquent, le DD utilise son propre temps CPU et sa mémoire pour effectuer des opérations pour le compte des VMs. Les ordonnanceurs actuels ne considèrent pas ce temps CPU dans le partage du CPU entre les VMs. Ce temps CPU se décompose comme suit :

— T_1 : dans l'hyperviseur pour gérer les interruptions matérielles.

— T_2 : dans l'OS du DD pour gérer les interruptions virtuelles et transférer les requêtes d'E/S entre le pilote réel et le backend.

— T_3 : dans le backend pour le multiplexage et la transmission/réception des requêtes d'E/S vers/depuis l'interface réseau. T_3 comprend le temps CPU nécessaire pour les transferts des droits sur les pages partagés (initiés avec des hypercalls).

Le temps CPU consommé dans le DD est significatif et dépend de l'activité dans les VMs. Dans les ordonnanceurs actuels, le temps CPU consommé par le DD pour les VMs n'est pas facturé aux VMs. Ceci est donc autant problématique autant pour les clients (prévisibilité des performances), que pour les fournisseurs de cloud (gaspillage de ressources). Deux approches sont possibles pour la configu-

ration de la capacité CPU à allouer au DD. Soit la capacité CPU du DD est limitée, soit le DD a accès à toute la ressource CPU de la PM.

— *Capacité limitée pour le DD (des ressources CPU bien définies pour le DD).* Dans ce cas de figure, l'isolation des performances des VMs peut être mise en cause, car l'activité d'une VM réalisant des opérations d'E/S peut influer sur les performances des autres VMs faisant de même [68]. En effet, si la capacité allouée au DD ne permet pas de satisfaire toutes les VMs (la charge CPU générée dans le DD est supérieure à la capacité CPU allouée), alors l'activité d'une VM aura un impact sur les autres, ce qui conduit au problème d'imprévisibilité des performances.

— *capacité illimitée pour le DD (le DD peut utiliser toutes les ressources de la PM).*Ce cas de figure revient à remettre un chèque en blanc au client, ce qui va en l'encontre de l'esprit du cloud qui est le *pay-as-you-go*.

Dans le cadre de cette thèse, nous nous intéressons principalement au CPU utilisé par le DD et l'étude de la mémoire est fera l'objet d'un futur travail.

4.1.2 L'hétérogénéité des infrastructures comme source d'imprévisibilité des performances

La deuxième source du problème d'imprévisibilité des performances dans le Cloud est l'hétérogénéité des infrastructures. Dans un IaaS, les VMs peuvent migrer entre des PMs hétérogènes à cause des mécanismes tels que la consolidation des serveurs. La quantité de ressources allouées à une VM à sa création peut-être exprimée par une valeur relative [59] (fraction de la ressource totale) ou une valeur absolue (valeur indépendante de la nature ou de la capacité totale de la ressource). L'allocation des ressources fondée sur des valeurs relatives peut causer des problèmes, car la capacité de traitement de la ressource peut varier suite à une migration sur des PMs hétérogènes. Par exemple, 50% d'un pCPU dépend de la nature (capacité) du pCPU. Nous constatons que : la mémoire, le disque et le réseau sont généralement alloués avec des valeurs absolues d'allocation. Mais ceci n'est pas le cas pour le CPU qui en général est alloué avec des valeurs relatives. Cette allocation du CPU axée sur des valeurs relatives peut conduire à deux situations :

— *soit le client est pénalisé (non-respect du SLA).* Ceci arrive quand une VM est déplacée sur des PMs hétérogènes et que le CPU de la PM de départ plus puissant que le CPU de la PM de réception. Ainsi, la VM se verra exécutée sur une machine moins puissante que l'initiale, et aura donc des performances moins bonnes.

— *soit le fournisseur est pénalisé (gaspillage de ressources).* Ceci arrive lorsque la quantité de ressources allouées par l'ordonnanceur à une VM d'un client est supérieure à celle convenue (migration de la VM d'un CPU moins puissant à un CPU plus puissant).

Dans les deux situations précédentes, le fournisseur ne garantit plus la qualité de service convenue avec le Client, ce qui conduit au problème d'imprévisibilité des performances.

4.2 État de l'art

4.2.1 Partage du DD entre les VMs et l'imprévisibilité des performances

Plusieurs travaux se sont penchés sur la gestion des opérations d'E/S dans les environnements virtualisés et l'impact que cela peut avoir sur la prévisibilité des performances. [72, 26, 67, 27] proposent des solutions qui peuvent être implémentées au niveau applicatif dans le DD pour répondre à ce problème d'imprévisibilité. Il s'agit d'allouer à une VM un débit pour les opérations d'E/S et de s'assurer que la VM aura toujours accès à ce débit. En garantissant ce débit, les VMs auront toujours les mêmes performances ce qui évite le problème d'imprévisibilité. La principale limite à cette approche vient du fait que le débit fourni à une VM peut être fortement affecté par l'activité CPU en cours dans le DD. Ainsi, allouer un débit à une VM ne sert à rien si le DD à une charge CPU intense et n'est pas en mesure de fournir le débit alloué à la VM. Ces solutions allouent une quantité de ressource CPU limitée au DD. [45] est un travail qui présente des observations très intéressantes sur le partage du DD avec les différents ordonnanceurs de l'hyperviseur Xen, et propose une solution pour le problème d'imprévisibilité des performances. Les auteurs de l'article proposent d'utiliser Xenmon [46] (un outil de monitoring) afin de surveiller la charge CPU générée par une VM dans le DD et par la suite bloquer les opérations d'E/S de la VM au cas où le total de sa charge CPU (celle générée dans le DD plus celle utilisée par la VM) dépasserait la capacité CPU allouée. Bloquer les opérations d'E/S d'une VM revient à supprimer dans le DD toute requête d'E/S qui lui est destinée. Ce travail présente de nombreuses limites. La première vient du fait que la solution qu'ils proposent ne s'intéresse qu'au réseau et ne traite pas du disque. La seconde limite vient de l'action de bloquer les opérations d'E/S d'une VM en supprimant les paquets qui lui sont destinés. Cette action peut avoir des effets néfastes sur certaines applications dans la VM. Une troisième limite provient de l'utilisation permanente de

Xenmon comme outil de monitoring. Ce dernier consomme de la ressource CPU donc induit un overhead important.

4.2.2 L'hétérogénéité dans le Cloud et l'imprévisibilité des performances

De nombreux travaux de recherche ont étudié le problème de l'hétérogénéité dans les centres d'hébergement [66, 33]. Nous nous intéressons à l'hétérogénéité des infrastructures. [52] évalue l'impact de considérer un centre d'hébergement comme homogène lorsqu'il est hétérogène. Il propose une métrique pour exprimer la sensibilité d'une application face à l'hétérogénéité. Selon leurs documentations respectives, Amazon EC2 [1] et Microsoft Azure [20] évitent le problème d'imprévisibilité des performances dû à l'hétérogénéité en dédiant le même type de matériel au même type de VM. Mais cette approche a des limites, car ces fournisseurs ne peuvent garantir cette correspondance lorsque les VMs sont migrées sur des sites géographiques différents qui ne disposent pas toujours du même matériel. De plus, forcer cette correspondance type VM/machine limite les migrations possibles durant une opération de consolidation des serveurs. En ce qui concerne Rackspace [14], aucune information n'est fournie sur la gestion du problème d'hétérogénéité. Nous avons pu constater par des évaluations simples que ce fournisseur ne gère pas bien l'hétérogénéité, car pour un type de VM, nous obtenons des performances différentes pour une application CPU intensive à différentes heures de la journée.

Dans les sections précédentes, nous avons décrit comment un mauvais partage du DD et l'hétérogénéité des infrastructures dans le Cloud sont à l'origine de l'imprévisibilité des performances. Nous avons également décrit quelques travaux qui se sont attaqués à ce problème. Dans la suite du chapitre, nous présentons nos deux contributions sur le problème d'imprévisibilité des performances. La première contribution s'intéresse au partage du DD, et propose un système de facturation du temps CPU utilisé par le DD pour les opérations d'E/S aux VMs bénéficiaires. La deuxième contribution s'intéresse à l'allocation dans les Clouds hétérogènes et propose une approche d'allocation permettant de garantir la capacité de calcul allouée à une VM quelle que soit l'hétérogénéité des CPUs dans l'infrastructure.

4.3 Facturation du temps CPU du DD aux VMs bénéficiaires

Dans cette section, nous décrivons notre solution qui surmonte le problème de partage du DD entre les VMs décrit dans la section 4.1. Cette solution, se met en œuvre au niveau de l'ordonnanceur de l'hyperviseur. Bien que la solution soit relativement facile à décrire, sa mise en œuvre doit faire face aux défis suivants :

— *Précision.* Comment comptabiliser avec précision le temps CPU utilisé par le DD pour chaque VM sachant que l'hyperviseur et le DD sont partagés entre les VMs ?

— *Overhead.* L'overhead induit par la solution doit être négligeable.

— *Généricité.* La solution doit nécessiter moins de modifications possibles dans l'OS invité.

4.3.1 Description générale

Nous proposons une solution qui prend en compte tous les défis énumérés ci-dessus. Cette solution repose principalement sur du calibrage. Elle se résume comme suit. Tout d'abord, le fournisseur mesure pour les différentes PMs du Iaas le temps CPU (noté $t = T_1 + T_2 + T_3$) nécessaire pour gérer chaque type de requête d'E/S (voir la section 4.3.2). Cette étape est effectuée une fois et les mesures sont mises à la disposition de l'ordonnanceur de l'hyperviseur. Ensuite, le DD est modifié afin de compter et collecter le nombre de requêtes d'E/S réalisées par chaque VM (appelé nb_{req}) en tenant compte du type de la requête. Cette modification est située dans le backend, qui est le lieu idéal pour suivre toutes les requêtes d'E/S. Par la suite, le DD envoie périodiquement les informations collectées à l'ordonnanceur. Ceci est réalisé à l'aide d'un hypercall (*gnttab_batch_copy*) qui est utilisé par le backend du DD. Cet hypercall est invoqué à chaque opération d'E/S. L'utilisation d'un hypercall existant évite l'introduction d'overhead. Lorsque l'ordonnanceur reçoit les informations collectées dans le DD, il calcule (en fonction de t, obtenu pendant la phase de calibrage) le temps CPU utilisé par le DD pour le compte de chaque VM : $nb_{req} \times t$. Ce temps CPU est alors facturé à la VM dans l'ordonnanceur de l'hyperviseur. Les sections suivantes donnent plus de détails sur le calibrage et la mise en œuvre de notre solution dans l'ordonnanceur Credit de Xen.

4.3.2 Le calibrage

La phase de calibrage consiste à évaluer $t = T_1 + T_2 + T_3$, le temps CPU nécessaire pour traiter chaque type de requête d'E/S dans le DD. Pour ce faire, nous disposons de sondes à la fois à l'entrée et à la sortie de chaque composant du DD pour le traitement des requêtes d'E/S. Nous décidons d'ignorer T_1 car il est très petit (conclusion similaire dans [71]). Les évaluations dans la section 4.3.4 confirment qu'ignorer T_1 est acceptable et n'a aucun impact sur la solution ce qui implique que $t \approx T_2 + T_3 = T_{DD}$. Nous avons implémenté un ensemble de micro-benchmarks [1] pour évaluer avec précision T_2 et T_3. Les sections suivantes présentent la méthodologie et les résultats obtenus du calibrage. Pour chaque type de requêtes d'E/S (disque ou réseau), nous considérons tous les facteurs qui pourraient avoir un impact sur le temps CPU de traitement des requêtes d'E/S dans le DD. Les facteurs les plus importants sont les suivants :

— *le type de l'opération d'E/S.* Les opérations disques et réseaux ne nécessitent pas le même temps CPU pour le traitement dans le DD,

— *le mode utilisé par le DD pour le routage des requêtes aux VMs.* Pour chaque périphérique, l'hyperviseur fournit plusieurs configurations possibles pour le routage des requêtes d'E/S au niveau du DD,

— *la taille de la requête d'E/S.* Les requêtes d'E/S n'ont pas toutes la même taille.

4.3.2.1 Le calibrage du réseau

Nous avons implémenté en *C* une application Client/Serveur pour déterminer le coût de chaque opération réseau au niveau du DD. Le client et le serveur sont dans le même réseau local.

Configurations

Xen fournit 3 modes de configuration pour le réseau dans le DD. Il s'agit du bridging, le routing et le NAT. Bridging est la configuration la plus utilisée. Le chemin pris par les paquets entre le pilote réel et le backend définit la différence entre les différents modes. Le routing et le NAT utilisent un chemin très similaire alors qu'il est différent pour le bridging. Comme on peut le voir dans la figure 4.2 (a), le routing nécessite plus de temps CPU que le bridging. Sauf si indication contraire, dans la suite du document nous utilisons le mode bridging.

1. Notons que, une fois l'hyperviseur et le DD sont modifiés (pour inclure les sondes), le calibrage ne nécessite pas beaucoup de temps (environ 5 minutes).

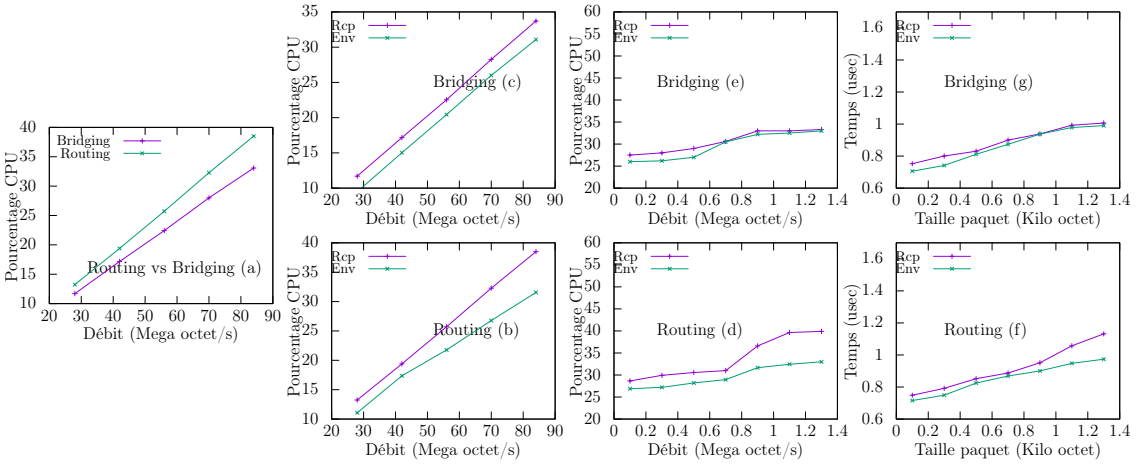

FIGURE 4.2 – Calibrage des opérations réseaux

Sens de la requête d'E/S

La transmission de paquets (lorsqu'une VM envoie un paquet) et réception de paquets (lorsqu'une VM reçoit un paquet) ne suivent pas le même chemin dans le backend [71]. Par conséquent, le temps de traitement nécessaire pour gérer un paquet réseau dans le DD dépend de son sens. Comme l'illustre la figure 4.2, la transmission est moins coûteuse que la réception. Cette différence vient du backend qui effectue notamment un hypercall supplémentaire lors de la réception d'un paquet comparé à la transmission.

Taille du paquet

Comme indiqué dans les 2 courbes (g) et (f) de la figure 4.2, le coût de la gestion d'un paquet réseau dans le DD varie avec sa taille. Cette dépendance à la taille du paquet provient du pilote réel qui réalise des copies de données.

4.3.2.2 Le calibrage du disque

Nous avons utilisé l'application *dd* (une application de copie) fournie par Linux pour le calibrage des opérations du disque. Comme dans le cas du réseau, il est possible d'avoir plusieurs modes de configuration pour la gestion du disque : *tab*, *phy* et *qdisk*. Dans le cadre de notre travail, nous avons étudié le mode *phy* qui est le plus utilisé. Nous avons mesuré les temps CPU en fonction de la taille des requêtes et de la nature des opérations (lecture ou écriture). La figure 4.3 présente les résultats du calibrage.

Configurations

Xen fournit différents modes de configuration du DD pour gérer les opérations

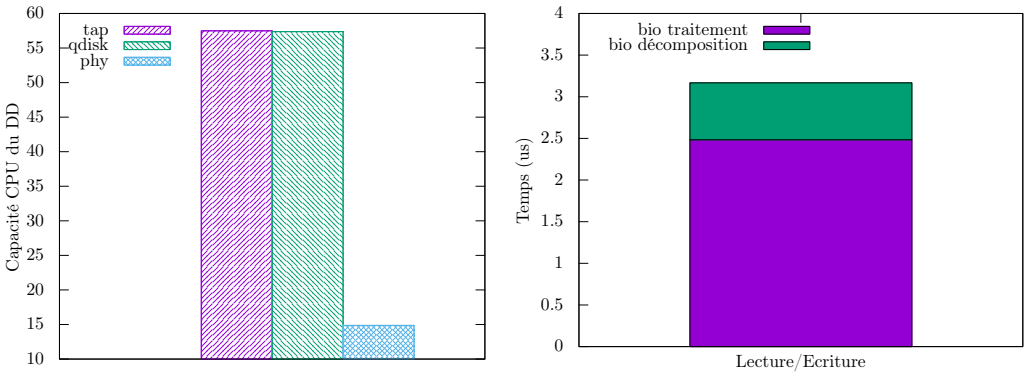

FIGURE 4.3 – Calibrage des opérations disques

disques qui sont : *tap*, *qdisk* et *phy*. Le mode de configuration influe sur le temps de traitement utilisé par le DD. *tap* ainsi que *qdisk* nécessitent beaucoup plus de capacité CPU pour le traitement dans le DD que *phy* (voir la figure 4.3.2.2 de gauche). Nous utilisons *phy* dans la suite de la section.

Nature de la requête d'E/S

Les opérations sur le disque sont soit une lecture, soit une écriture. Contrairement au réseau, ils sont toujours initiés par la VM. Selon Linux et le code source du backend, leur traitement suit le même chemin et la lecture comme l'écriture nécessitent le même temps de traitement dans le DD. Par conséquent, notre implémentation ne distingue pas le type d'opération du disque.

Taille du paquet

Une requête vers le disque est limitée par la taille de la page mémoire (par exemple, 4KB). Une requête qui arrive au backend est fragmentée en plusieurs sous-structures de données *bio*, l'unité de traitement du noyau Linux. Tous les bios sont de la même taille. Ainsi, notre calibrage est effectué à la granularité d'un bio. Par conséquent, nous évaluons d'une part le temps utilisé pour la construction d'un bio (notons $T_{bio\_dcomposition}$) à partir d'une requête de disque. D'autre part, nous évaluons le temps nécessaire pour le traitement d'un bio (notons $T_{bio\_traitement}$). La figure 4.3 de droite montre les résultats du calibrage pour notre environnement expérimental.

4.3.3 Implémentation

Cette section présente l'implémentation de notre solution dans l'OS du DD (Linux version 3.13.11.7) et l'hyperviseur Xen (version 4.2.0). Cette implémentation n'est pas intrusive pour les VMs, car elle nécessite uniquement la modifi-

cation du backend (qui est un module de l'OS Linux) et de l'hyperviseur. En ce qui concerne le premier, de nouvelles structures de données ont été introduites pour stocker des informations sur le nombre de requêtes d'E/S traitées par les VMs. Bénéficiant de l'existence de l'hypercall *gnttab_batch_copy* utilisé à la fin de chaque opération d'E/S, le backend envoie le contenu de ses structures de données à l'hyperviseur. Ceci se fait périodiquement après un nombre défini de requêtes traitées. À partir des résultats du calibrage, l'ordonnanceur de l'hyperviseur connaît le coût (temps CPU) de traitement d'un type de requêtes d'E/S en fonction de la taille et pour différentes configurations (routing, NAT et bridging pour le réseau et qdisk, tap et phy pour le disque). L'ordonnanceur utilise donc toutes ses informations pour déterminer le temps CPU utilisé par le DD pour les VMs, et ajouter ce temps CPU lors du calcul du crédit utilisé par les VMs. Nous avons modifié la structure de données des VMs dans l'hyperviseur (*struct_domain*) afin d'y ajouter le temps CPU utilisé par le DD pour le compte d'une VM : net_{debt} et $disk_{debt}$. Ces variables contiennent les dettes de temps CPU que la VM doit rembourser.

4.3.4 Évaluations

Dans cette section, nous présentons les résultats obtenus de l'évaluation de notre solution. L'évaluation comporte plusieurs cas d'étude : utilisation des micro-benchmarks et utilisation des benchmarks plus complexes. Nous évaluons les aspects suivants de notre solution : l'overhead et l'efficacité en termes de prévisibilité des performances des applications.

4.3.4.1 L'environnement expérimental

Les évaluations ont été réalisées dans un IaaS privé constitué de machines HP ayant chacune un processeur Intel Core i7-3770 et 8 Giga-octets de mémoire. Pour toutes les évaluations, le DD est limité à ne pouvoir utiliser que 30% d'un pCPU (qui représente sa capacité). Nous avons alloué cette capacité au DD pour montrer que deux VMs réalisant des opérations d'E/S peuvent se perturber lorsque le DD à une capacité de calcul limité. Les VMs sont configurées avec un seul vCPU et ils sont ordonnancés sur des pCPUs différents de celui utilisé par le DD.

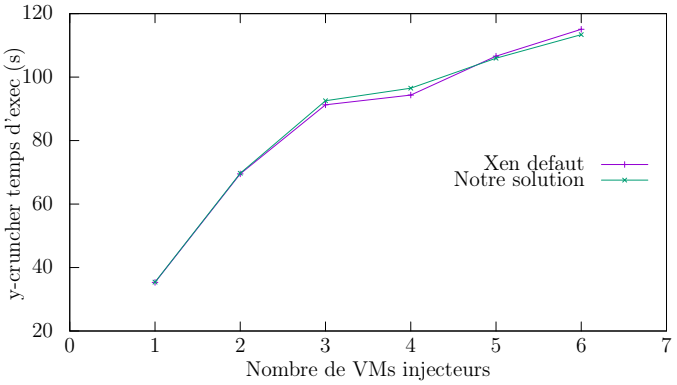

FIGURE 4.4 – Évaluation de l'overhead.

4.3.4.2 Overhead

La figure 4.4 présente les résultats des expériences que nous avons effectuées pour évaluer l'overhead de notre solution. Nous exécutons une VM témoin (notée $v_{witness}$) hébergeant une application (y-cruncher [22] CPU intensive). $V_{witness}$ est configurée avec un seul vCPU fixé sur le même pCPU que le DD (ayant également un seul vCPU). Le DD et $v_{witness}$ ont accès à l'ensemble de la capacité de ce pCPU. La PM héberge également un ensemble de VMs (appelées injecteurs) dont le nombre varie pendant les expériences (pour augmenter le trafic dans le DD). Chaque VM exécute la même application Web qui est wordpress. Les expériences ont été réalisées dans deux contextes. Le premier contexte est avec le système Xen par défaut, c'est la référence (la base). Le deuxième contexte est notre solution avec le mécanisme de facturation des dettes aux VMs désactivé (ce qui veut dire que nous comptons les requêtes d'E/S dans le DD, nous transmettons les infos à l'hyperviseur mais nous ne facturons pas aux VMs dans l'ordonnanceur de l'hyperviseur). La figure 4.4 montre que notre solution n'introduit pas d'overhead puisque la performance de $v_{witness}$ est la même dans les deux contextes, de ce fait, les mécanismes que nous avons ajoutés dans le DD et l'hyperviseur n'introduisent aucun overhead.

4.3.4.3 La précision de notre solution

Évaluation avec des micro-benchmark

Deux benchmarks ont été utilisés : (1) wordpress [21] pour le réseau et (2) la commande linux *dd* pour le disque. Durant les évaluations, les VMs sont limitées à n'utiliser que 30% d'un pCPU (capacité 30) quand l'évaluation concerne le réseau et 15% (capacité 15) lorsqu'il s'agit du disque. L'évaluation se fait dans deux contextes : avec notre solution et avec le système Xen par défaut. Les résultats ob-

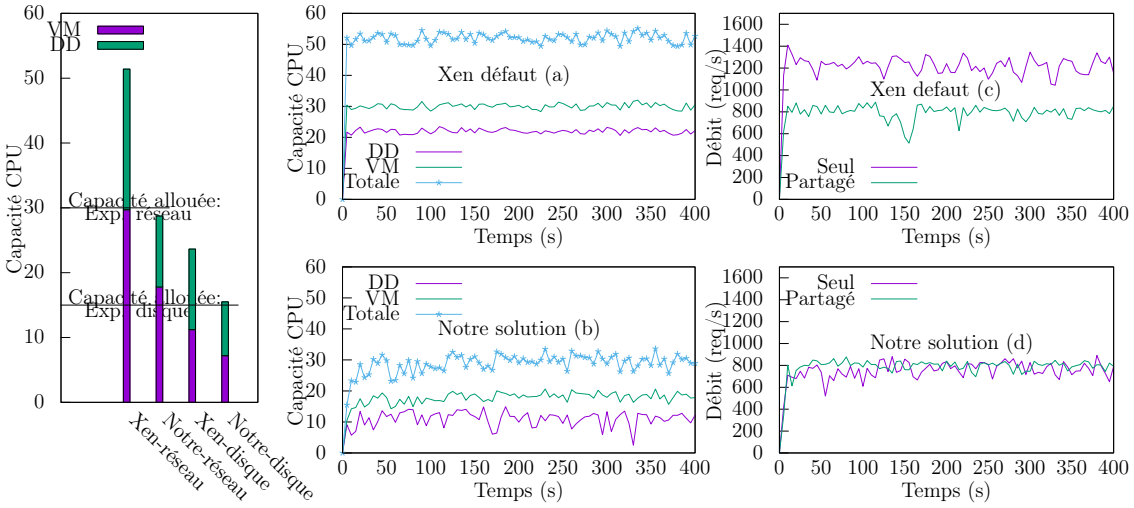

FIGURE 4.5 – Précision de notre solution en utilisant des micro-benchmarks

tenus montrent que notre solution garantit que la charge totale de CPU générée par une VM (la charge de la VM plus la charge générée dans le DD) est égale à la quantité allouée (30% ou 15% d'un pCPU suivant si c'est le disque ou le réseau) à la VM. Ceci n'est pas le cas avec le système Xen par défaut. S'assurer que la capacité CPU allouée est toujours respectée nous permet de garantir la prévisibilité des performances des applications. La courbe tout à droite de la figure 4.5 présente les charges CPUs obtenues pour le disque et le réseau. Nous pouvons constater qu'avec notre solution, la charge totale CPU d'une VM est égale à la capacité allouée (30% ou 15% d'un pCPU) ce qui n'est pas le cas avec Xen par défaut. Les courbes (a) et (b) de la figure 4.5 mettent l'accent sur la charge CPU durant l'évaluation du réseau. Ces courbes montrent que la charge CPU de la VM avec notre solution est toujours limitée à la capacité allouée durant toute l'évaluation ce qui n'est pas le cas avec Xen par défaut. Les deux courbes (c) et (d) de la figure 4.5 se focalisent sur le réseau. Elles présentent les résultats obtenus sur la prévisibilité des performances lorsque deux VMs utilisent simultanément le DD. La courbe (c) montre que ce problème de prévisibilité existe dans Xen par défaut lorsque deux VMs partagent le DD. En effet, le débit de la VM est de 1200 req/s quand elle est seule à utiliser le DD, ce débit passe à 800 req/s quand elle est co-localisée avec une autre. La courbe (d) montre les résultats de la même évaluation quand notre solution est utilisée. Nous pouvons constater que la VM garde la même performance, environ 800 req/s que la VM soit seule ou pas. Ce débit est celui que doit fournir la VM avec la capacité qui lui a été allouée. Notre implémentation évite la saturation du DD vu que la charge CPU qu'elle génère est facturée aux VMs.

Évaluation avec des benchmark complexe

Cette évaluation montre la justesse de notre solution avec un ensemble de bench-

marks fourni par SPECvirt sc2010 [17] (SPECvirt). Ce dernier est un benchmark de référence qui est très utilisé par les fournisseurs de cloud pour l'évaluation de leur plateforme. Il est composé de trois benchmarks : SPECweb2005 (application web), SPECjAppserver2004 (application JEE), et SPECmail2008 (application mail). Chaque benchmark dispose d'un indicateur de performance : le temps moyen de réponse des requêtes pour SPECweb2005 et SPECmail2008, et le JIOPS (java Operation Per Seconds) pour SPECjAppserver2004. Nous avons exécuté chaque benchmark dans une VM mono-vCPU avec une capacité de 30% d'un CPU. Nous avons expérimenté deux scénarios de co-localisation : (1) chaque benchmark est exécuté seul sur la machine physique, et (2) tous les benchmarks sont exécutés simultanément. La figure 4.6 présente les résultats de cette expérimentation. Les courbes du dessus donnent les performances des applications, tandis que celles du bas présentent les charges CPU du DD et des VMs durant l'expérimentation. Nous constatons que SPECjAppserver2004 génère très peu d'activités dans le DD (la barre verte de la figure 4.6(c) du bas est plus petite que sur les figures. 4.6 (a) et (b) du bas). Ceci s'explique par le fait que SPECjAppserver2004 ne réalise pas beaucoup d'opérations d'E/S en comparaison avec SPECweb2005 et SPECmail2008. De ce fait, notre solution et le system de Xen par défaut conduisent pratiquement au même résultat pour SPECjAppserver2004 quand ce dernier est exécuté seul ou avec les autres benchmarks. A contrario, Xen par défaut n'assure pas la prévisibilité des performances autant pour SPECweb2005 (la deuxième et la quatrième barres vertes de la figure 4.6 (b) du bas n'ont pas la même hauteur), que pour SPECmail2008 (la deuxième et la quatrième barres vertes de la figure 4.6 (a) du bas n'ont pas la même hauteur). Avec une marge d'erreur négligeable, notre solution garantit la prévisibilité des performances (les temps de réponse illustrés par la première et la troisième barres des figures. 4.6 (a) et (b) de haut ont la même hauteur). Notre solution permet d'obtenir de tels résultats car elle limite chaque VM à n'utiliser que les capacités CPU qui leurs ont été allouées. Ceci peut être vu sur la figure 4.6, car la charge CPU totale représentée par la première et la troisième barres des figures 4.6(a) et (b) du bas sont près de 30% (la capacité allouée).

4.3.5 Synthèse

Dans les systèmes de virtualisation, les temps CPU utilisés par le DD au profit des VMs ne sont pas facturés aux VMs. Ceci peut avoir un impact sur la prédictibilité des performances dans un Cloud. Ainsi, nous venons de décrire une extension des systèmes de virtualisation qui permet de comptabiliser les temps CPU utilisés par le DD au profit des VMs et de les facturer aux VMs. Cette extension a été implémentée sous Xen et une évaluation en a été faite.

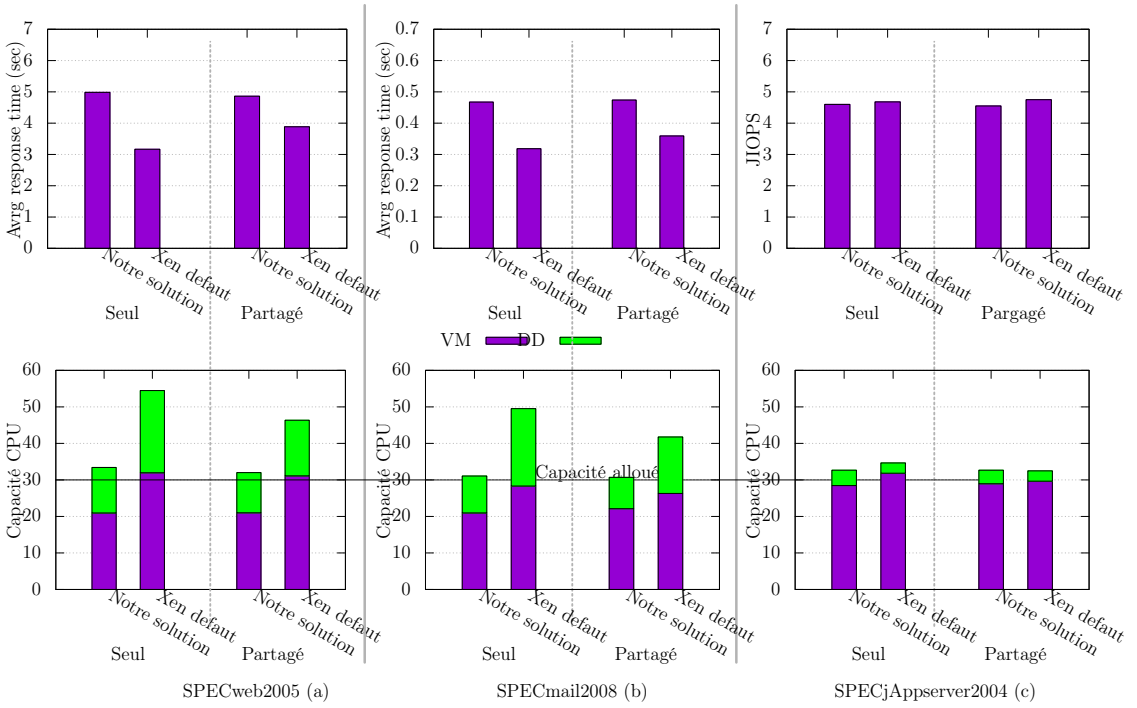

FIGURE 4.6 – Précision de notre solution avec SPECvirt comme benchmark

4.4 Allocation absolue dans un Cloud hétérogène

Dans cette section, nous abordons le problème d'imprévisibilité des performances sous le prisme de l'hétérogénéité des infrastructures dans le Cloud. En général dans les IaaS, les vCPUs sont fixés sur des pCPUs. Notre travail se situe dans ce contexte. Le fournisseur présente la capacité de calcul d'un vCPU comme celle d'un pCPU ou d'une fraction. Ceci devient très ambigu dans un Cloud hétérogène, car une VM peut se voir migrée sur des PMs différentes avec des CPUs différents. Ceci conduit au problème d'imprévisibilité des performances, car la capacité de calcul d'un vCPU dépendra du pCPU sur lequel il s'exécute. Nous présentons dans cette section un modèle d'allocation visant à résoudre ce problème imprévisibilité provenant de l'hétérogénéité des infrastructures. La présentation de ce modèle se fera en répondant aux deux questions suivantes :

— Comment clairement définir la capacité de calcul d'un vCPU ?

— Comment garantir cette capacité de calcul d'un vCPU ?

La suite de la section est organisée comme suit : nous introduisons tout d'abord notre capacité de calcul d'un vCPU et par la suite nous décrivons comment cette capacité est garantie dans un contexte hétérogène.

4.4.1 Description générale

Dans notre système d'allocation, la capacité de calcul d'un vCPU est présentée comme celle d'un CPU spécifique, que nous nommons p_{ref}, disponible dans le IaaS. p_{ref} est choisi une seule fois par le fournisseur et doit être le CPU avec la capacité de calcul la plus basse du IaaS (En cas d'ajout d'un CPU moins puissant que p_{ref}, il doit être changé). A chaque migration de VM sur des PMs hétérogènes avec des pCPUs différents de p_{ref}, seulement un pourcentage d'un pCPU sera attribué à un vCPU de la VM. Notre solution se déroule en deux phases : nous avons une phase de calibrage, et une phase de calcul du pourcentage de pCPU à allouer à un vCPU en cas de migration. Cette deuxième phase se repose essentiellement sur les résultats du calibrage.

Le Calibrage

Notons app une application CPU intensive mono-thread et $ExecutionTime(app, p)$, le temps d'exécution de notre application lorsqu'elle s'exécute exclusivement sur un pCPU p. Notre système garantit la capacité de calcul d'un vCPU si et seulement si, pour un vCPU v,
$ExecutionTime(app, p)$=$ExecutionTime(app, p_{ref})$ quel que soit le pCPU sur lequel s'exécute le vCPU. Nous définissons le coefficient de proportionnalité (noté $coef(p)$) entre p_{ref} et tout autre pCPU p comme suit :

$$coef(p) = \frac{ExecutionTime(app, p_{ref})}{ExecutionTime(app, p)} \tag{4.1}$$

Le calibrage consiste à déterminer ce coefficient de proportionnalité sur tous les PMs du IaaS. Le coefficient est calculé une seule fois par le fournisseur pour tous les CPUs du IaaS. En ce qui concerne app, elle doit être une application CPU intensive et utilisant peu de mémoire car notre système s'adresse au problème d'allocation du CPU.

Garantir l'allocation

A chaque migration de VM sur une PM avec un CPU autre que p_{ref}, seulement $\frac{1}{coef(p)}$ % du CPU de la nouvelle PM sera alloué aux vCPUs de la VM. Avec cette contrainte, nous sommes certains que la capacité d'un vCPU sera respectée quelque soit la PM sur laquelle il s'exécute.

4.4.2 Implémentation

L'ordonnanceur Credit de Xen fonctionne sur un système de crédits attribués aux VMs. Ces crédits sont consommés par la VM à chaque fois qu'elle utilise le

DELL (p_0)	Intel Core 2 Duo CPU E7300 @ 2.66GHz	Ubuntu 12.04
HP (p_1)	Intel Core i7-3770 CPU @ 3.40GHz	-

TABLE 4.1 – Machines expérimentales

CPU. L'extension que nous avons proposé dans l'ordonnanceur consiste à adapter la consommation des crédits en fonction du pCPU sur lequel s'exécute le vCPU de la VM. Ainsi, si un vCPU s'exécute sur un pCPU autre que p_{ref}, le calcul du crédit consommé (voir section 2.4.1) pour ce vCPU se fait non seulement à partir du temps CPU utilisé, mais également du coefficient de proportionnalité de ce CPU (le calcul est pondéré). Avec cette approche, le calcul du crédit dans l'ordonnanceur se fait toujours comme si le vCPU s'exécute sur p_{ref}. Par conséquent, la capacité de calcul pour un vCPU sera toujours égale à celle de p_{ref} quel que soit le CPU sur lequel s'exécute le vCPU.

4.4.3 Évaluations

Cette section présente les résultats d'évaluation de notre modèle d'allocation. Nous avons évalué la capacité de notre solution à garantir une capacité de calcul d'un vCPU définie, en d'autres termes à garantir une prévisibilité des performances des applications dans le Cloud.

4.4.3.1 L'environnement expérimental et les benchmarks

Toutes les expériences dans cette section sont faites sur des machines physiques hétérogènes, donc les caractéristiques sont présentées dans le tableau 4.1. Nous avons évalué notre solution avec SPECcpu2006 [15] qui est une suite de benchmarks CPU mono-thread.

4.4.3.2 La précision de notre solution

Méthodologie

La capacité de calcul définie pour un vCPU est respectée si et seulement si une application sur ce vCPU a le même temps d'exécution quel que soit le pCPU de la PM exécutant le vCPU. L'évaluation de nos modèles consiste donc à observer les variations sur les performances des applications lorsqu'une VM est migrée entre des PMs hétérogènes. Pour montrer la criticité de ce problème, nous avons éga-

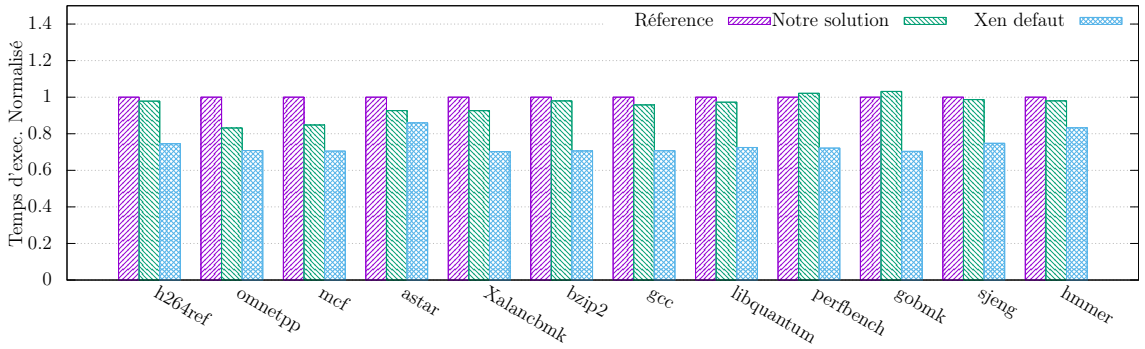

FIGURE 4.7 – Évaluation avec SPEC CPU2006.

lement évalué cette variation avec l'ordonnanceur Credit de Xen (Xen defaut). Pour résumer, notre évaluation s'est faite comme suit : nous avons réalisé plusieurs exécutions des benchmarks sur des PMs hétérogènes avec notre système d'allocation et avec l'ordonnanceur Credit de Xen. La première exécution se fait sur le processeur de référence, cette exécution donne les performances qui sont attendues (référence). Ensuite nous exécutons nos benchmarks sur des PMs n'ayant pas le processeur de référence avec notre solution (Solution) et avec l'ordonnanceur Credit de Xen (Xen defaut). Par la suite, nous analysons les différences entre les différentes exécutions. Dans le cadre de notre évaluation, le CPU de référence est p_0 car il est le moins puissant. Ainsi, nous définissons la capacité d'un vCPU comme celle de p_0.

Résultats avec SPEC

SPEC CPU est exécuté dans une VM à un vCPU. La figure 4.7 contient les histogrammes présentant les temps d'exécution normalisés par rapport à l'exécution référence. Nous pouvons observer que notre système d'allocation conduit à un temps d'exécution proche du temps de référence après plusieurs exécutions. Ceci n'est pas le cas pour le système Xen par défaut (Xen defaut). Ce dernier conduit à une variation importante du temps d'exécution lorsque la VM ne s'exécute pas sur p_0. Cependant, nous constatons que les résultats obtenus avec notre solution présentent une variation un peu importante avec quelques benchmarks. Cet écart est expliqué par le fait que les benchmarks de SPECcpu2006 ne sont pas seulement CPU intensifs, mais que certains sont également mémoire intensifs ([65]) et les PMs sur lesquelles nous réalisons les expériences ont des mémoires hétérogènes. Rappelons que notre solution ne traite que l'allocation du CPU, et non celle de la mémoire. Ainsi, dans cette expérience, notre solution résout le problème du CPU, et la différence vient de la mémoire hétérogène.

4.4.4 Synthèse

Cette contribution s'attaque au problème d'imprévisibilité des performances dûe à l'hétérogénéité des infrastructures dans le Cloud. Nous avons proposé un modèle d'allocation et son implémentation dans le système de virtualisation Xen. Cette solution repose sur un pCPU de référence (p_{ref}) comme l'unité d'allocation des vCPUs. Par la suite, en se fondant sur un coefficient de proportionnalité entre les autres pCPUs et (p_{ref}), l'ordonnanceur est capable d'allouer strictement à un vCPU la capacité de calcul correspondant à (p_{ref}) en cas de migration sur des PMs hétérogènes.

4.5 Synthèse

Dans ce chapitre, nous avons introduit le problème d'imprévisibilité des performances dans les Cloud. L'imprévisibilité des performances est la conséquence immédiate de l'incapacité du fournisseur à garantir le SLA. Nous parlons du problème d'imprévisibilité des performances si les performances d'une application s'exécutant dans une VM du Iaas varient dans le temps. En d'autres termes, le fournisseur ne garantit pas l'isolation des performances. Nous avons identifié les deux importantes causes de ce problème dans le Cloud : un mauvais partage des ressources entre les différentes VMs et l'hétérogénéité des infrastructures dans les centres d'hébergement. Nous avons présenté deux contributions pour résoudre le problème d'imprévisibilité des performances. La première contribution s'intéresse au partage du DD, et propose un système de facturation du temps CPU utilisé par le DD pour les opérations d'E/S aux VMs bénéficiaires. La deuxième contribution s'intéresse à l'allocation dans les Clouds hétérogènes et propose une approche d'allocation permettant de garantir la capacité de calcul allouée à une VM quelle que soit l'hétérogénéité des CPUs dans l'infrastructure.

Chapitre 5

Conclusion et Perspectives

Contents

5.1 Conclusion

Depuis plusieurs années, nous assistons à l'essor du Cloud Computing. Le principe fondateur de cette pratique est d'externaliser la gestion des services informatiques des entreprises dans des centres d'hébergement gérés par des entreprises tierces. Cette externalisation a pour principal avantage une réduction des coûts pour l'entreprise cliente. Le Cloud computing utilise principalement la virtualisation comme principale technologie. La virtualisation permet la mutualisation et le partage des ressources entre différents clients. Les fournisseurs de Cloud utilisent la virtualisation pour implémenter l'isolation aussi bien sur le plan de la sécurité que des performances. Malgré tous les avantages qu'apporte la virtualisation au Cloud Computing, elle porte son lot de défis, notamment liés à l'efficacité dans le partage des ressources entre les VMs et également sur la mise en œuvre de l'isolation des performances entre les VMs.

Nos premières contributions dans cette thèse portent sur l'efficacité de l'ordonnancement dans les environnements virtualisés. Dans un système virtualisé, l'ordonnanceur de l'hyperviseur se charge de multiplexer le CPU entre tous les vCPUs des VMs. Le partage du CPU entre tous les vCPUs est à l'origine d'une discontinuité dans l'exécution des vCPUs sur les pCPUs. Ainsi, un vCPU n'a pas accès à tout instant au pCPU, mais uniquement pour une durée de temps appelée quantum. Cette discontinuité a un important impact sur certains mécanismes de l'OS invité tels que la gestion des interruptions et les Spinlock. En effet, ces mécanismes sont fondés sur l'hypothèse fondamentale selon laquelle un OS est capable de réquisitionner le CPU à tout instant pour réaliser des opérations importantes. Cette hypothèse n'est plus vraie dans les environnements virtualisés, car c'est l'hyperviseur qui gère le matériel et non les OS invités. Cela conduit à une croissance de la latence de traitement des opérations d'Entrée/Sortie dans le cas de la gestion des interruptions, et également au problème de LHP et LWP en ce qui concerne les Spinlocks. Pour résoudre ces problèmes, nous avons proposé deux contributions. La première s'implémente exclusivement au niveau de l'hyperviseur et la seconde nécessite une collaboration entre les OS invités et l'hyperviseur. La première contribution est un nouvel ordonnanceur appelé *AQL_Sched*. Nous avons constaté qu'une bonne valeur de quantum dans l'ordonnanceur de l'hyperviseur permet de résoudre ces problèmes dus à la discontinuité. Fort de ce constat, nous avons conçu *AQL_Shed*, un ordonnanceur qui ordonnance le même type de vCPU sur un pool de pCPUs dédié, en utilisant le meilleur quantum (le quantum qui conduit ce type à sa meilleure performance). Pour ce faire, l'ordonnanceur comprend trois fonctionnalités importantes ; (1) un système de reconnaissance du type d'un vCPU (vTRS) ; (2) l'identification du meilleur quantum par le calibrage ; (3) un clustering qui est le regroupement des vCPUs en fonction

de leur affinité de quantum. Nous avons évalué notre ordonnanceur et les résultats démontrent qu'il réduit l'effet des problèmes provenant de la discontinuité. Notre seconde contribution sur l'efficacité de l'ordonnancement propose une collaboration entre l'OS invité et l'hyperviseur. Nous avons présenté I-Spinlock, une nouvelle primitive de Spinlock dans l'OS invité qui ne permet à un thread d'acquérir un ticket que si le quantum residuel de son vCPU est suffisant pour entrer et quitter la section critique sans être préempté par l'hyperviseur. I-Spinlock repose sur les mécanismes de mémoire partagés pour informer les vCPUs de la durée restante de leur quantum afin que l'OS invité puisse déterminer si le thread peut prendre un ticke ou non. Nous avons évalué I-Spinlock, et l'évaluation démontre que I-Spinlock évite problèmes de LHP et LWP dans les OS invités.

Les contributions suivantes de notre thèse portent sur l'imprévisibilité des performances dans un Cloud. Dans un IaaS, les fournisseurs de Cloud mettent à la disposition des clients des VMs avec des capacités définies de ressources. Ces capacités de ressources allouées à la VM correspondent à un accord (qualité de service) entre le fournisseur et le client, ce dernier s'attend à un respect de la qualité de service. L'imprévisibilité des performances est l'incapacité du fournisseur de Cloud à garantir cette qualité de service à tout instant. Nous avons recensé deux causes à l'imprévisibilité des performances qui sont (i) un mauvais partage des ressources et (ii) la migration des VMs sur des machines hétérogènes. En ce qui concerne (i), l'hyperviseur dispose de plusieurs composants responsables du partage des ressources matérielles. Le DD est un composant de l'hypersieur chargé de la gestion des périphériques d'Entrée/Sortie. Nous avons constaté que les VMs qui réalisent des opérations d'E/S génèrent une charge CPU importante dans le DD et que cette charge n'est pas facturée à ces VMs dans l'ordonnanceur de l'hyperviseur. Dans le cas d'un DD configuré avec une capacité CPU limitée, cela peut conduire à un problème d'imprévisibilité. En effet, le DD peut ne pas avoir assez de capacité CPU pour satisfaire les besoins de toutes les VMs. Ainsi l'activité d'une VM réalisant des opérations d'E/S pourra impacter les autres. Pour ce qui est de (ii), nous avons constaté qu'en général, l'allocation du CPU dans un IaaS repose sur des valeurs relatives (donc la capacité dépend de la capacité totale de la ressource). Cette situation est problématique car les VMs sont amenées à être migrées sur des PMs hétérogènes à cause de la consolidation des serveurs. Ainsi, la capacité CPU alouée à une VM à sa création ne sera pas toujours respectée car celle ci sera migrée sur une PM différente de sa PM initiale (donc CPU différent). Pour répondre à ces problèmes, nous avons proposé deux contributions. La première est une extension de l'ordonnanceur de l'hyperviseur qui permet de facturer le temps CPU utilisé par le DD pour des VMs à ces dernières. Cette solution repose principalement sur du calibrage. Elle se résume comme suit. Tout d'abord, le fournisseur mesure pour les différents PMs du Iaas le temps CPU nécessaire pour gérer chaque type de requête d'E/S. En-

suite, le DD est modifié afin de compter et collecter le nombre de requêtes d'E/S réalisées par chaque VM. Par la suite, le DD envoie périodiquement les informations collectées à l'ordonnanceur qui les facture aux VMs concernées. Nous avons implémenté cette solution et nous l'avons évaluée. L'évaluation a démontré que notre solution résout le problème d'imprévisibilité des performances dû au partage du DD. La seconde contribution sur l'imprévisibilité des performances est un système d'allocation. Nous avons présenté un système d'allocation dans lequel la capacité de calcul d'un vCPU est présentée comme celle d'un pCPU spécifique, nommé p_{ref}, disponible dans le IaaS. A chaque migration de VM sur des PMs hétérogènes avec des pCPUs différents de p_{ref}, seul un pourcentage d'un pCPU sera attribué à un vCPU de la VM. Nous avons implémenté et évalué notre système d'allocation. Les évaluations démontrent que notre solution permet de garantir la prévisibilité des performances malgré l'hétérogénéité des infrastructures.

5.2 Perspectives

Tout au long de la réalisation de cette thèse, nous avons identifié différents axes potentiels d'amélioration de nos contributions. Ces axes peuvent être classés en deux catégories : les travaux réalisables dans un futur proche, et ceux réalisables dans un futur plus lointain compte tenu de leur ampleur.

5.2.1 Perspectives à court terme

— *Calibrage automatique avec AQL_Sched*. Notre ordonnanceur *AQL_Sched* repose sur une phase de calibrage off-line pour déterminer la meilleure valeur de quantum pour un type d'application. Une amélioration serait de permettre un calibrage en temps réel. En d'autres termes, déterminer le meilleur quantum pour un type d'application durant l'exécution de cette dernière et d'utiliser automatiquement cette valeur déterminée dans l'ordonnanceur de l'hyperviseur.

— *Gestion des E/S avec I-Spinlock*. Notre primitive de spinlock repose entièrement sur la disponibilité et la validité du temps restant avant la fin du quantum d'un vCPU. Cette hypothèse peut être mise à mal si une interruption intervient au niveau de l'hyperviseur et que ce dernier préempte un vCPU avant la fin du quantum. I-Spinlock doit évoluer afin de prendre en compte ce scénario.

— *La prise en compte de la mémoire dans la gestion du DD*. Notre solution actuelle sur le partage du DD entre les VMs porte sur la ressource CPU, pourtant la

ressource mémoire du DD est également utilisée. Notre analyse doit s'étendre sur la mémoire et réaliser une étude sur la gestion de la mémoire dans le DD.

— *Allocation absolue sur des systèmes asymétriques*. Le système d'allocation que nous avons proposé pour pallier le problème d'hétérogénéité dans le Cloud ne s'applique qu'aux architectures de CPU symétrique (SMP). Il serait intéressant, d'étendre ce travail à de nouvelles architectures asymétriques dans lesquelles l'hétérogénéité est ramenée à la granularité du CPU. Dans un système asymétrique, les pCPUs du même CPU sont hétérogènes ce qui complexifie un peu plus le système.

5.2.2 Perspectives à long terme

Dans cette thèse, nous avons présenté quatre contributions qui permettent d'améliorer les performances dans un environnement virtualisé. Chaque contribution améliorant un aspect de la virtualisation. Notre objectif à long terme serait de tester la compatibilité de toutes ces solutions et d'évaluer le gain total venant de ces contributions.

Bibliography

[1] Amazon Elastic Compute Cloud. http ://aws.amazon.com/ec2/. Consulté le 2017-05-3.

[2] Apps.Gov. https ://www.apps.gov/. Consulté le 2017-05-3.

[3] AWS Elastic Beanstalk. http ://aws.amazon.com/elasticbeanstalk/. Consulté le 2017-05-3.

[4] Ebizzy. 'http ://sourceforge.net/projects/ebizzy/. Consulté le 2016-05-3.

[5] Google App Engine Cloud Platform. https ://cloud.google.com/products/. Consulté le 2017-05-3.

[6] Google Apps for Business. http ://www.google.com/apps/. Consulté le 2017-05-3.

[7] Google Compute Engine. https ://cloud.google.com/products/compute-engine. Consulté le 2017-05-3.

[8] Google Documents. https ://docs.google.com. Consulté le 2017-05-3.

[9] IBM SmartCloud Application Services. http ://www.ibm.com/cloud-computing/us/en/paas.html. Consulté le 2017-05-3.

[10] Kernbench. http ://ck.kolivas.org/apps/kernbench/kernbench-0.50/. Consulté le 2015-05-3.

[11] Microsoft Office 365. http ://office.microsoft.com/. Consulté le 2017-05-3.

[12] para-virtualized spinlock. https ://lwn.net/Articles/556141/. Consulté le 2017-05-3.

[13] Pbzip2. 'http ://compression.ca/pbzip2/. Consulté le 2016-05-3.

[14] Rackspace. http ://www.rackspace.com/. Consulté le 2014-05-3.

[15] SPECCPU2006. https ://www.spec.org/cpu2006/. Consulté le 2015-05-3.

[16] SPECmail2009. https ://www.spec.org/mail2009/press/release.html. Consulté le 2015-05-3.

[17] specvirt. https ://www.spec.org/virt sc2010/. Consulté le 2015-01-3.

[18] SPECWeb2009. https ://www.spec.org/web2009/. Consulté le 2015-05-3.

[19] Ticket Spinlock. https ://lwn.net/Articles/267968/,. Consulté le 2017-05-3.

[20] Windows Azure : Microsoft's Cloud Platform. http ://www.windowsazure.com/en-us/. Consulté le 2017-05-3.

[21] Wordpress. https ://wordpress.org/plugins/benchmark/. Consulté le 2015-05-3.

[22] ycruncher. A Multi-Threaded Pi-Program. http ://www.numberworld.org/y-cruncher/. Consulté le 2015-05-3.

[23] Ahn, J., Park, C. H., and Huh, J. (2014). Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 394–405, Washington, DC, USA. IEEE Computer Society.

[24] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A View of Cloud Computing. *Commun. ACM*, 53(4) :50–58.

[25] Avi Kivity, Yaniv Kamay, D. L. U. L. A. L. (2007). KVM : the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*.

[26] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. (2011). Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA. ACM.

[27] Ballani, H., Jang, K., Karagiannis, T., Kim, C., Gunawardena, D., and O'Shea, G. (2013). Chatty Tenants and the Cloud Network Sharing Problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 171–184, Berkeley, CA, USA. USENIX Association.

[28] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA. ACM.

[29] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. (2009). The Multikernel : A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA. ACM.

[30] Benzina, H. (2012). *Enforcing virtualized systems security. (Renforcement de la sécurité des systèmes virtualisés)*. PhD thesis, École normale supérieure de Cachan, France.

[31] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite : Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[32] Buyya, R., Broberg, J., and Goscinski, A. M. (2011). *Cloud Computing Principles and Paradigms*. Wiley Publishing.

[33] Canini, M., Jovanović, V., Venzano, D., Novaković, D., and Kostić, D. (2011). Online Testing of Federated and Heterogeneous Distributed Systems. *SIGCOMM Comput. Commun. Rev.*, 41(4) :434–435.

[34] Cheng, L. and Wang, C.-L. (2012). vBalance : Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 2 :1–2 :14, New York, NY, USA. ACM.

[35] Cherkasova, L., Gupta, D., and Vahdat, A. (2007). Comparison of the Three CPU Schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2) :42–51.

[36] Das Kamhout, Greg Bunce, C. P. (2010). Implementing On-Demand Services Inside the Intel IT Private Cloud.

[37] Ding, X., Gibbons, P. B., Kozuch, M. A., and Shan, J. (2014). Gleaner : Mitigating the Blocked-waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 73–84, Berkeley, CA, USA. USENIX Association.

[38] Drepper, U. (2007). What Every Programmer Should Know About Memory.

[39] Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., and Veidenbaum, A. V. (2012). Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 389–400, Washington, DC, USA. IEEE Computer Society.

[40] Fraser, K., H, S., Neugebauer, R., Pratt, I., and Williamson, M. (2004). Safe hardware access with the Xen virtual machine monitor. In *In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*.

[41] Friebel, T. How to Deal with Lock-Holder Preemption. Technical report.

[42] Fuchi, K., Tanaka, H., Manago, Y., and Yuba, T. (1969). A Program Simulator by Partial Interpretation. In *Proceedings of the Second Symposium on Operating Systems Principles*, SOSP '69, pages 97–104, New York, NY, USA. ACM.

[43] Goldberg, R. P. (1973). Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, New York, NY, USA. ACM.

[44] Goth, G. (2007). Virtualization : Old Technology Offers Huge New Potential. 8(2) : ? ?– ? ?

[45] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. (2006). Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA. Springer-Verlag New York, Inc.

[46] Gupta, D., Gardner, R., and Cherkasova, L. (2005). XenMon : QoS Monitoring and Performance Profiling tool. Technical report.

[47] Intel Corporation (2009). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US.

[48] Jaeger, T., Sailer, R., and Sreenivasan, Y. (2007). Managing the Risk of Covert Information Flows in Virtual Machine Systems. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 81–90, New York, NY, USA. ACM.

[49] Kashyap, S., Min, C., and Kim, T. (2016). Opportunistic Spinlocks : Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Oper. Syst. Rev.*, 50(1) :9–16.

[50] Li-jie Jin, Vijay Machiraju, A. S. (2002). Analysis on Service Level Agreement of Web Services. Technical report, HP Laboratories.

[51] Love, R. (2010). *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition.

[52] Mars, J. and Tang, L. (2013). Whare-map : Heterogeneity in "Homogeneous" Warehouse-scale Computers. *SIGARCH Comput. Archit. News*, 41(3) :619–630.

[53] Marsh, B. D., Scott, M. L., Leblanc, T. J., and Markatos, E. P. (1991). First-Class User-Level Threads. In *In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121.

[54] McKenney, P. E. (2004). *Exploiting Deferred Destruction : An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University. Available : http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed October 15, 2004].

[55] Mell, P. and Grance, T. (2009). The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*, 53(6) :50.

[56] Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1) :21–65.

[57] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W. (2005). Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA. ACM.

[58] Microsoft (2011). Microsoft's Top 10 Business Practices for Environmentally Sustainable Data Centers.

[59] Nicolo', A. (2004). Efficiency and truthfulness with Leontief preferences. A note on two-agent, two-good economies. *Review of Economic Design*, 8(4) :373–382.

[60] Nikolaev, R. and Back, G. (2011). Perfctr-Xen : A Framework for Performance Counter Virtualization. *SIGPLAN Not.*, 46(7) :15–26.

[61] Nowak, A. and Bitzes, G. (2014). The overhead of profiling using PMU hardware counters.

[62] Ousterhout, J. (1982). Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30.

[63] Ouyang, J. and Lange, J. R. (2013). Preemptable Ticket Spinlocks : Improving Consolidated Performance in the Cloud. *SIGPLAN Not.*, 48(7) :191–200.

[64] Parekh, A. K. and Gallager, R. G. (1993). A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks : The Single-node Case. *IEEE/ACM Trans. Netw.*, 1(3) :344–357.

[65] Phansalkar, A., Joshi, A., and John, L. K. (2007). Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Computer Architecture News*, 35(1) :69–76.

[66] Phothilimthana, P. M., Ansel, J., Ragan-Kelley, J., and Amarasinghe, S. (2013). Portable Performance on Heterogeneous Architectures. *SIGARCH Comput. Archit. News*, 41(1) :431–444.

[67] Popa, L., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. (2011). FairCloud : Sharing the Network in Cloud Computing. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 22 :1–22 :6, New York, NY, USA. ACM.

[68] Pu, X., Liu, L., Mei, Y., Sivathanu, S., Koh, Y., Pu, C., and Cao, Y. (2013). Who Is Your Neighbor : Net I/O Performance Interference in Virtualized Clouds. *IEEE Trans. Serv. Comput.*, 6(3) :314–329.

[69] Qureshi, M. K. and Patt, Y. N. (2006). Utility-Based Cache Partitioning : A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA. IEEE Computer Society.

[70] Sahoo, J., Mohapatra, S., and Lath, R. (2010). Virtualization : A Survey on Concepts, Taxonomy and Associated Security Issues. In *Proceedings of the 2010 Second International Conference on Computer and Network Technology*, ICCNT '10, pages 222–226, Washington, DC, USA. IEEE Computer Society.

[71] Santos, J. R., Turner, Y., Janakiraman, G., and Pratt, I. (2008). Bridging the Gap Between Software and Hardware Techniques for I/O Virtualization. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA. USENIX Association.

[72] Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. (2010). Runtime Measurements in the Cloud : Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1-2) :460–471.

[73] Song, Z., Zhang, X., and Eriksson, C. (2015). Data Center Energy and Cost Saving Evaluation. *Energy Procedia*, 75 :1255 – 1260. Clean, Efficient and Affordable Energy for a Sustainable Future : The 7th International Conference on Applied Energy (ICAE2015).

[74] Sukwong, O. and Kim, H. S. (2011). Is Co-scheduling Too Expensive for SMP VMs ? In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 257–272, New York, NY, USA. ACM.

[75] T, R. K. (2013). Virtual Cpu Scheduling Techniques for Kernel Based Virtual Machine (Kvm). In *2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–6.

[76] Tang, L., Mars, J., and Soffa, M. L. (2011a). Contentiousness vs. Sensitivity : Improving Contention Aware Runtime Systems on Multicore Architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA. ACM.

[77] Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L. (2011b). The Impact of Memory Subsystem Resource Sharing on Datacenter Applications.

In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 283–294, New York, NY, USA. ACM.

[78] Tchana, A., Bui, V. Q. B., Teabe, B., Nitu, V., and Hagimont, D. (2016). Mitigating performance unpredictability in the IaaS using the Kyoto principle. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 6.

[79] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. (2005). Intel Virtualization Technology. *Computer*, 38(5) :48–56.

[80] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. (2004). Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA. USENIX Association.

[81] van Doorn, L. (2006). Hardware Virtualization Trends. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 45–45, New York, NY, USA. ACM.

[82] Velte, A. and Velte, T. (2010). *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[83] Vmware (2015). vSphere Virtual Machine Administration. In .

[84] Wells, P. M., Chakraborty, K., and Sohi, G. S. (2006). Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 124–133, New York, NY, USA. ACM.

[85] Weng, C., Liu, Q., Yu, L., and Li, M. (2011). Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 239–250, New York, NY, USA. ACM.

[86] Wu, S., Xie, Z., Chen, H., Di, S., Zhao, X., and Jin, H. (2016). Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 343–352.

[87] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. F. (2013). Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 233–240, Washington, DC, USA. IEEE Computer Society.

[88] Xu, C., Gamage, S., Lu, H., Kompella, R., and Xu, D. (2013). vTurbo : Accelerating Virtual Machine I/O Processing Using Designated Turbo-sliced Core. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 243–254, Berkeley, CA, USA. USENIX Association.

[89] Xu, C., Gamage, S., Rao, P. N., Kangarlou, A., Kompella, R. R., and Xu, D. (2012). vSlicer : Latency-aware Virtual Machine Scheduling via Differentiated-frequency CPU Slicing. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 3–14, New York, NY, USA. ACM.

[90] Yadav, A. and Apte, V. (2016). Dynamic Server Consolidation Algorithms : A Profit Model for Evaluation and an Improvement. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pages 31 :1–31 :4, New York, NY, USA. ACM.

Liste des tableaux

Table des figures