

UNIVERSITE DE YAOUNDE I  
\*\*\*\*\*  
ECOLE NATIONALE SUPERIEURE  
POLYTECHNIQUE  
\*\*\*\*\*  
DEPARTEMENT DE GENIE  
INFORMATIQUE



UNIVERSITY OF YAOUNDE I  
\*\*\*\*\*  
NATIONAL ADVANCED SCHOOL  
OF ENGINEERING  
\*\*\*\*\*  
DEPARTMENT OF COMPUTER  
ENGINEERING

---

## TECHNIQUE D'ESTIMATION DU WORKING SET BASEE SUR LE PML (PAGE MODIFICATION LOGGING)

---

### MÉMOIRE DE FIN D'ÉTUDES

Présenté et soutenu par :

**CÉLESTINE STELLA N'DONGA BITCHEBE**

En vue de l'obtention du :

**DIPLÔME D'INGÉNIEUR DE CONCEPTION, OPTION GÉNIE  
INFORMATIQUE**

Sous la supervision de :

**ALAIN TCHANA, PROFESSEUR, UNIVERSITÉ NICE SOPHIA  
ANTIPOLIS**

**LAURENT REVEILLERE, PROFESSEUR, INSTITUT NATIONAL  
POLYTECHNIQUE DE BORDEAUX**

Devant le jury composé de :

**Président : CLAUDE TANGHA, PROFESSEUR,  
UNIVERSITÉ DE YAOUNDÉ I**

**Rapporteur 1 : THOMAS BOUETOU BOUETOU,  
PROFESSEUR, UNIVERSITÉ DE YAOUNDÉ I**

**Rapporteur 2 : ALAIN TCHANA, PROFESSEUR,  
UNIVERSITÉ NICE SOPHIA ANTIPOLIS**

**Examineur : THOMAS DJOTIO NDIE, MAÎTRE DE  
CONFÉRENCES, UNIVERSITÉ DE YAOUNDÉ I**

**Année académique 2017-2018  
Le 14 septembre 2018**



---

**TECHNIQUE D'ESTIMATION DU WORKING  
SET BASEE SUR LE PML (PAGE  
MODIFICATION LOGGING)**

---

**MÉMOIRE DE FIN D'ÉTUDES**

Présenté et soutenu par :

**CÉLESTINE STELLA N'DONGA BITCHEBE**

En vue de l'obtention du :

**DIPLÔME D'INGÉNIEUR DE CONCEPTION, OPTION GÉNIE  
INFORMATIQUE**

Année académique 2017-2018  
Le 14 septembre 2018



## DÉDICACE

A ma famille

A decorative flourish consisting of two symmetrical, flowing lines that curve upwards and then downwards, framing the text "A ma famille". Below this, there is a horizontal line with a series of small, repeating loops or ripples.

## REMERCIEMENTS

Ma gratitude va à l'endroit de tous ceux qui ont contribué à mon éducation, à ma formation et à la réalisation de ce travail :

- **Pr Claude TANGHA**, pour l'honneur qu'il me fait en acceptant de présider ce jury.
- **Pr Thomas BOUETOU BOUETOU**, pour son encadrement, son dévouement dans l'enseignement et la direction de notre département de génie informatique.
- **Pr Alain TCHANA** et **Pr Laurent REVEILLERE**, pour leur disponibilité, leurs directives, leurs conseils, leur suivi, ainsi que pour l'opportunité qu'ils m'ont offerte d'effectuer ce stage.
- **Pr Thomas DJOTIO NDIE**, pour l'honneur qu'il me fait en acceptant d'examiner scientifiquement ce travail.
- La Direction de l'Institut de Recherche en Informatique de Toulouse, pour m'avoir permis d'effectuer mon stage dans le sein de son illustre institution.
- Le corps enseignant de l'École Nationale Supérieure Polytechnique de Yaoundé, en particulier celui du Département de Génie Informatique, pour la formation et le savoir qu'il s'est dévoué à me transmettre. Je remercie tout spécialement **Pr Thomas BOUETOU BOUETOU**, **Pr Thomas DJOTIO NDIE**, **Dr Bernabé BATCHAKUI** et **Dr Georges Edouard KOUAMOU**.
- Les membres de l'équipe SEPIA de l'IRIT, équipe dans laquelle j'ai effectué ce stage : **Boris TEABE**, **Djob MVONDO**, **Lavoisier WAPET**, **Kevin JIOKENG**, **Vlad NITU**, **Grégoire TODESCHI**, **Mathieu BACOU** pour leur collaboration et l'aide qu'ils m'ont apportée pendant le déroulement de ce stage.

Je ne saurais manquer d'adresser ma reconnaissance envers :

- **Mes parents**, **M. Désiré BITCHEBE** et **Mme Cécile FETNGO**, pour leur soutien, leurs conseils et leur perpétuelle présence pour moi.
- **Mes soeurs**, pour leur présence, leurs encouragements et leur amour.



- **Stéphane TJOMB**, pour son accompagnement chaque jour, son soutien inébranlable, son écoute et son affection.
- La promotion **GI2018**, avec qui nous avons partagé des connaissances et des idées. En particulier **Peterson YUHALA**, avec qui nous avons vécu cette expérience de stagiaires, **Junior UM-GWET**, **Rahimatou DAOUDA**, **Ange MEKOULOU** et **Raoul DZOUKOU**.
- Tous mes amis, qui m'encouragent et me soutiennent chaque jour.



# TABLE DES MATIÈRES

<b>Dédicace</b>	<b>i</b>
<b>Remerciements</b>	<b>ii</b>
<b>Sigles et abréviations</b>	<b>vii</b>
<b>Résumé</b>	<b>ix</b>
<b>Abstract</b>	<b>x</b>
<b>Tableaux</b>	<b>xi</b>
<b>Codes</b>	<b>xi</b>
<b>Figures</b>	<b>xii</b>
<b>Introduction</b>	<b>1</b>
<b>1 Concepts généraux</b>	<b>5</b>
1.1 Généralités sur la Virtualisation . . . . .	6
1.1.1 Définition . . . . .	6
1.1.2 Techniques de virtualisation . . . . .	7
1.1.3 Systèmes de virtualisation . . . . .	10
1.1.4 Avantages de la virtualisation . . . . .	11
1.2 Virtualisation de la mémoire . . . . .	12
1.2.1 Environnement natif (sans virtualisation) . . . . .	12

1.2.2	Environnement virtualisé . . . . .	13
1.3	Page Modification Logging (PML) . . . . .	17
1.3.1	Intel VT-x . . . . .	17
1.3.2	Description du PML . . . . .	18
<b>2</b>	<b>Etat de l'art sur l'estimation du working set</b>	<b>21</b>
2.1	Enoncé du problème . . . . .	22
2.1.1	Allocation de mémoire à la demande . . . . .	22
2.1.2	Les critères d'estimation du WSS . . . . .	24
2.2	Les techniques existantes . . . . .	24
2.2.1	Self-ballooning . . . . .	25
2.2.2	Zballoond . . . . .	26
2.2.3	VMware . . . . .	27
2.2.4	Geiger . . . . .	29
2.2.5	Hypervisor Exclusive Cache . . . . .	30
2.2.6	Dynamic MPA ballooning . . . . .	31
<b>3</b>	<b>Contribution : Technique d'estimation basée sur une architecture améliorée du PML</b>	<b>33</b>
3.1	Contribution sur le plan architectural . . . . .	34
3.1.1	Design actuel du PML . . . . .	34
3.1.2	Proposition d'un design amélioré pour l'estimation du WSS . . . . .	38
3.2	Algorithmique d'estimation . . . . .	42
3.3	Avantages de la solution proposée . . . . .	44
<b>4</b>	<b>Implémentation et Evaluations de la technique élaborée</b>	<b>46</b>
4.1	Implémentation . . . . .	47
4.1.1	Environnement d'implémentation . . . . .	47
4.1.2	Détails d'implémentation . . . . .	49
4.2	Évaluations . . . . .	55
4.2.1	Environnement d'évaluation . . . . .	55

---

4.2.2	Expérimentation 1 : charge synthétique manipulant un working set constant . . . . .	56
4.2.3	Expérimentation 2 : charges synthétiques manipulant un working set variable (en escaliers) . . . . .	57
<b>Conclusion</b>		<b>60</b>
<b>Références</b>		<b>62</b>
<b>Annexes</b>		<b>67</b>



## SIGLES ET ABRÉVIATIONS

<b>A/D</b>	Accessed and Dirty
<b>CPU</b>	Core Processing Unit
<b>DCs</b>	Datacenters
<b>ENSPY</b>	Ecole Nationale Supérieure Polytechnique de Yaoundé
<b>EPT</b>	Extended Page Table
<b>gla</b>	guest linear address
<b>gPA</b>	guest Physical Address
<b>gPT</b>	guest Page Table
<b>gVA</b>	guest Virtual Address
<b>hPA</b>	host Physical Address
<b>hPT</b>	host Page Table
<b>HVM</b>	Hardware Virtual Machine
<b>INPT</b>	Institut National Polytechnique de Toulouse
<b>Intel VT</b>	Intel Virtualization Technology
<b>IRIT</b>	Institut de Recherche en Informatique de Toulouse
<b>LaBRI</b>	Laboratoire Bordelais de Recherche en Informatique




<b>MMU</b>	Memory Management Unit
<b>NPT</b>	Nested Page Table
<b>OS</b>	Operating System
<b>PML</b>	Page Modification Logging
<b>SLA</b>	Service Level Agreement
<b>SLAT</b>	Second Level Address Translation
<b>TLB</b>	Translation Lookaside Buffer
<b>Vaddr</b>	Virtual address
<b>vCPU</b>	virtual Core Processing Unit
<b>VMCS</b>	Virtual Machine Control Strucutre
<b>VMM</b>	Virtual Machine Monitor
<b>VMs</b>	Machines Virtuelles
<b>VT-x</b>	Intel Virtual Technology Extensions
<b>WSS</b>	Working Set Size



## RÉSUMÉ

---

es dernières années ont vu la **virtualisation** s'imposer comme technologie de base dans les datacenters. Dans ces derniers, les utilisateurs ont tendance à surdimensionner les ressources de leurs applications, ce qui a pour conséquences : un gaspillage de la mémoire, un faible taux de consolidation et une proportionnalité énergétique faible. La mémoire étant la ressource critique, il serait avantageux d'allouer à une **machine virtuelle** la quantité exacte de mémoire dont elle a besoin à un instant donné. Il faudrait donc être capable d'anticiper sa demande (aussi bien à la hausse qu'à la baisse), i.e. de connaître à tout moment la taille de son **working set**. De nombreuses techniques d'estimation du working set existent déjà, basées sur des solutions dites logicielles, qui sont intrusives et/ou actives, et qui surchargent la machine virtuelle et/ou l'hyperviseur. Dans ce travail nous proposons une technique d'estimation qui permet de pallier les problèmes que posent les solutions logicielles. En effet notre approche est basée sur une technologie matérielle : le **PML** (Page Modification Logging). Ceci la laisse totalement externe à la machine virtuelle (non intrusive et non active) et de plus tous les traitements se font au niveau du **dom0** (pas de surcharge de la machine virtuelle ni de l'hyperviseur).

**Mots clés : Virtualisation, Machine virtuelle, Working Set, PML, dom0.**

---

## ABSTRACT

---

**I**n recent years *virtualization* has emerged as a core technology in the datacenters. In these datacenters, users tend to overdrive the resources of their applications, which leads to a waste of resources, low consolidation and low energy proportionality. Memory being the most critical resource, it would be advantageous to allocate to a *virtual machine*, the exact amount of memory it needs at a given moment. It is necessary so to be able to wait for his request, to know at any time the size of his *work set*. Many techniques of working set size estimation already exist, based on software solutions that are intrusive and / or active, and which overload the virtual machine and / or the hypervisor. In this work we propose a solution which makes it possible to overcome the problems that software solutions pose. Indeed, our approach is based on a feature called page modification logging (*PML*), which leaves it totally external to the virtual machine (non-intrusive and non-active) and all the processes are done at *dom0* level (no overloading of the virtual machine nor of the hypervisor).

**Keywords :** Virtualization, Virtual machine, Working Set, PML, dom0.

---

## TABLEAUX

1.1	Systèmes de virtualisation en fonction du type de virtualisation . . . . .	10
2.1	Tableau comparatif des techniques d'estimation du WSS existantes . . . . .	32
4.1	Caractéristiques de la machine physique dans les expérimentations . . . . .	55
4.2	Tableau comparatif des techniques d'estimation du WSS existantes . . . . .	60

## CODES

4.1	xl enable-log-dirty . . . . .	69
4.2	xl disable-log-dirty . . . . .	69
4.3	Méthode appelée générer une feuille L1 avant modification de la bitmap .	70
4.4	Méthode appelée pour générer un noeud L1 après modification de la bitmap	70
4.5	Méthode appelée lors de l'évènement <i>pml buffer full</i> avant modification . .	70
4.6	Portion de code modifiée dans de la méthode appelée lors de l'évènement <i>pml buffer full</i> . . . . .	73
4.7	xl collect-dirty-logs . . . . .	75
4.8	Fonction xc_domain_collect_dirty_logs . . . . .	75
4.9	Portion de code modifiée dans la méthode paging_log_dirty_op . . . . .	76
4.10	Script de collecte des logs . . . . .	78

## FIGURES

1.1	Architecture d'un système virtualisé . . . . .	6
1.2	Architecture Virtualisation niveau OS . . . . .	7
1.3	Architecture Noyau en espace utilisateur . . . . .	8
1.4	Architecture Virtualisation complète . . . . .	8
1.5	Architecture Para-virtualisation . . . . .	9
1.6	Architecture Virtualisation avec hyperviseur XEN . . . . .	11
1.7	Translation adresse virtuelle - adresse physique (environnement natif) . . . . .	12
1.8	Parcours table de page 1D (une dimension) . . . . .	13
1.9	Fonctionnement global de la TLB . . . . .	14
1.10	Mécanisme du shadow paging . . . . .	15
1.11	Mécanisme du nested paging . . . . .	16
2.1	Augmentation de la valeur du <i>Committed_As</i> avec l'instruction <i>malloc</i> même si ne correspondant pas à la valeur de la mémoire physique utilisée . . . . .	25
3.1	Architecture actuelle du PML . . . . .	34
3.2	Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Redirection des VMExits vers le dom0 . . . . .	39
3.3	Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Introduction d'un deuxième buffer <i>pml_log</i> . . . . .	40
3.4	Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Modification de la structure de données <i>bitmap</i> . . . . .	41
3.5	Variation de la taille de logs en fonction du temps . . . . .	42

---

3.6	Variation de la taille de logs en fonction du temps . . . . .	43
4.1	File d'appels des fonctions liées à l'activation du PML . . . . .	50
4.2	Radix tree représentant la bitmap . . . . .	51
4.3	Radix tree représentant la longmap . . . . .	52
4.4	Copie des logs de l'hyperviseur vers le dom0 . . . . .	53
4.5	Copie des logs de l'hyperviseur vers le dom0 : chaîne d'appel de fonctions . . .	54
4.6	Ensemble des charges de travail synthétiques . . . . .	55
4.7	Résultats d'évaluations de la première charge synthétique avec les techniques existantes . . . . .	56
4.8	Résultats d'évaluations de la première charge synthétique avec la technique implémentée . . . . .	57
4.9	Résultats d'évaluations des charges variables avec les techniques existantes .	58
4.10	Résultats d'évaluations des charges variables avec le PML . . . . .	59

# INTRODUCTION

## CONTEXTE

**L**e *cloud computing* consiste à externaliser les services d'une entreprise dans un centre d'hébergement géré par une autre entreprise (Amazon EC2 par exemple).

La *virtualisation*, qui a été rendue populaire grâce à cette pratique du *cloud computing*, s'impose comme la technologie de base dans les Datacenters (DCs), car elle favorise l'utilisation optimale des ressources (et donc une économie d'énergie) à travers la consolidation des machines virtuelles (VMs).

Néanmoins les gains observés dans les datacenters de production sont très faibles comparés à ceux attendus. En effet, le nombre de VMs consolidées dans un serveur est calculé en fonction des ressources réservées ou allouées à une machine virtuelle. Or il est connu que les ressources consommées par la VM varient au cours de son exécution. En outre les utilisateurs ont le souci de garder le même rendement tant sur le plan de la confidentialité (données) que des performances (applications), ce qui les pousse à surdimensionner les ressources de leurs applications. Autrement dit, ils réservent auprès des hébergeurs bien plus d'espace (mémoire, disque, CPU, etc.) qu'ils n'en ont besoin.

## PROBLEMATIQUE

Le fait que les utilisateurs surdimensionnent les ressources nécessaires peut avoir certaines conséquences :

- ☞ Un faible taux de consolidation et un rendement énergétique faible dans les dat-





acenters : il est à noter que la consommation électrique représente environ 50% - 70% des dépenses dans un datacenter[14].

☞ Un gaspillage de la mémoire.

Comme solution à ce problème de consolidation, des académies de chercheurs et des fournisseurs d'hyperviseurs ont proposé l'*overcommitment*<sup>1</sup>. Ce procédé nécessite une estimation fréquente des besoins actuels de chaque VM. Une mauvaise estimation pourrait impacter les applications des utilisateurs, particulièrement en ce qui concerne la mémoire car elle est considérée comme la ressource critique en terme de consolidation [10]. La quantité de mémoire actuelle dont une VM a besoin pour s'exécuter est appelée *Working Set Size (WSS)*.

Les techniques d'estimation du WSS existantes sont toutes basées sur une approche logicielle, ce qui induit certains inconvénients dont les plus notoires sont :

- ☞ L'imprécision (sous/sur-estimation)
- ☞ La surcharge (nécessitant beaucoup de ressources pour s'exécuter)
- ☞ L'intrusion (nécessitant la modification du système d'exploitation de la VM)

Notre problématique se formule donc tel qu'il suit : **Comment estimer le working set d'une machine virtuelle sans dégradations de performances ?**

## MOTIVATIONS ET OBJECTIFS

Etant données toutes les contraintes (présentées ci-haut) que posent les approches logicielles, nous avons pensé qu'une solution basée sur une approche matérielle répondrait à ces limites.

Depuis 2016, Intel en collaboration avec *VMWare* (fournisseur d'hyperviseurs), a commencé à produire des processeurs équipés d'une nouvelle technologie de virtualisation

---

<sup>1</sup>Procédé informatique qui consiste à allouer à une machine virtuelle plus de mémoire que ce dont dispose le système réel.



dénommée *Page Modification Logging (PML)*, qui donne à l'hyperviseur la possibilité de traquer efficacement les pages mémoire que la VM modifie durant son exécution[17]. Le PML a été introduit entre autres dans le but d'améliorer l'estimation du WSS, mais jusqu'ici aucune étude n'a été menée pour prouver son efficacité sur ce point. C'est la raison pour laquelle nous avons porté un intérêt particulier à cette fonctionnalité avec pour buts :

- ☛ D'étudier le mécanisme actuel du PML en mettant en exergue les limites qu'il présente dans le cadre de l'estimation du WSS.
- ☛ De proposer une nouvelle architecture mieux adaptée à ce problème.
- ☛ De présenter un algorithme de calcul du WSS qui s'appuie sur le nouveau design proposé.
- ☛ D'évaluer et de comparer notre technique d'estimation (qui est basée sur une approche matérielle) à celles existantes (qui sont basées sur des approches logicielles).

## ORGANISATION DU DOCUMENT

La suite du document s'organise tel qu'il suit :

- ☛ **Concepts généraux** : il s'agira ici de rappeler les définitions de certaines notions nécessaires à la compréhension du problème.
- ☛ **Etat de l'art** : ici nous allons présenter en détails les solutions existantes ainsi que les limites que pose chacune d'elles.
- ☛ **Contributions** : nous présenterons les détails de la solution que nous proposons.
- ☛ **Implémentation et Evaluations** : ce chapitre présentera quelques détails d'implémentation et les évaluations que nous avons effectuées pour valider l'intérêt de notre solution.





- ➡ **Conclusion et perspectives** : nous allons enfin conclure en présentant les améliorations envisageables.

## CONCEPTS GÉNÉRAUX

**D**ans ce chapitre nous introduisons ou rappelons les informations et concepts de base nécessaires à la compréhension du travail présenté dans ce document. Il s'agit des notions liées à la **virtualisation** et au **PML**.



## 1.1 Généralités sur la Virtualisation

### 1.1.1 Définition

La virtualisation est l'ensemble des techniques matérielles ou logicielles qui permettent de faire fonctionner simultanément sur une seule machine physique (machine hôte) plusieurs systèmes d'exploitation (*OS : Operating System*) appelés Machines Virtuelles (VMs) (*Virtual Machines, en anglais*). L'architecture d'un environnement virtualisé est présentée par la figure 1.1 suivante :

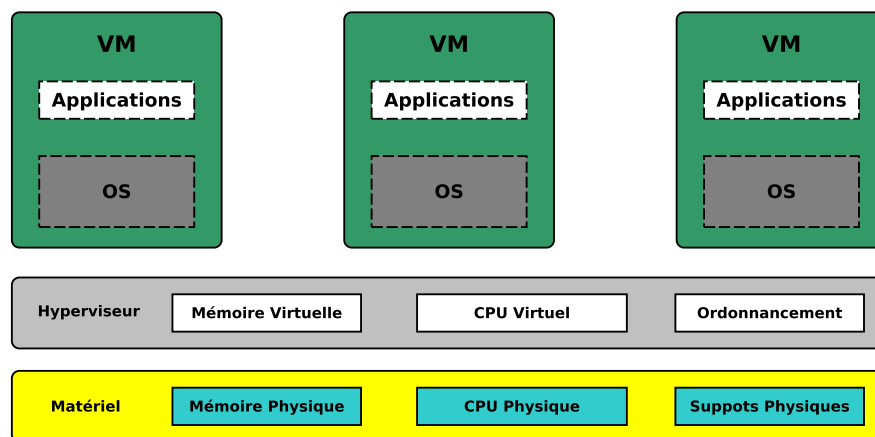


Figure 1.1: Architecture d'un système virtualisé

La plupart des serveurs (non virtualisés) utilisent moins de 15% de leurs capacités [30], ce qui favorise leur prolifération et la complexité de leur gestion. La virtualisation résout ces problèmes d'efficacité en permettant l'exécution de plusieurs systèmes d'exploitation sur un même serveur physique sous la forme de machines virtuelles, dont chacune peut accéder aux ressources de calcul du serveur sous-jacent. Chaque VM est une entité isolée, autonome et complètement indépendante des autres. Dans ces environnements virtualisés, un système de virtualisation, Virtual Machine Monitor (VMM), est responsable de la gestion de ces machines virtuelles et du partage des ressources entre elles. Il émule le matériel pour elles, et établit la communication entre elles et les périphériques (de la machine hôte). Il existe plusieurs techniques de virtualisation.





## 1.1.2 Techniques de virtualisation

En fonction de la position du système de virtualisation et des machines virtuelles par rapport au matériel, il existe cinq techniques de virtualisation:

### 1.1.2.1 Virtualisation niveau OS ou isolation

Cette technique de virtualisation permet d'isoler l'exécution des applications des zones d'exécution appelées contextes ou containers. Ici, l'OS hôte dispose des mécanismes pour construire des containers isolés (VMs). Ces derniers partagent le même OS (l'hôte). La VMM fait donc partie de l'OS hôte. Cette solution est très bonne en performance, mais les environnements virtualisés ne sont pas complètement isolés. La figure 1.2 présente l'architecture de cette technique de virtualisation.

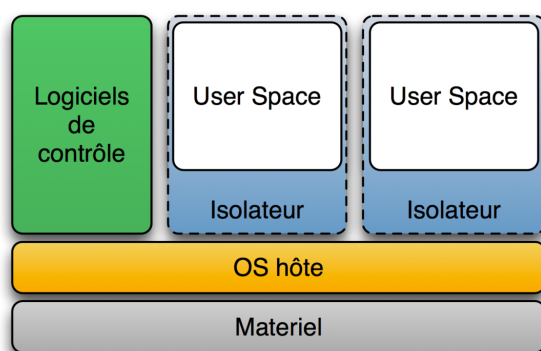


Figure 1.2: Architecture Virtualisation niveau OS

**Source : référence [27]**

**Exemples :** OpenVZ, BSD Jail, chroot, Linux-VServer, Linux containers.

### 1.1.2.2 Noyau en espace utilisateur

Un noyau en espace utilisateur (user-space) tourne comme une application en espace utilisateur de l'OS hôte. Le noyau user-space a donc son propre espace utilisateur dans lequel il contrôle ses applications. Cette solution est très peu performante, car deux noyaux sont empilés, l'isolation des environnements n'est pas gérée et l'indépendance

par rapport au système hôte est inexistante. Elle sert surtout au développement du noyau. La figure 1.3 présente l'architecture de cette technique de virtualisation.

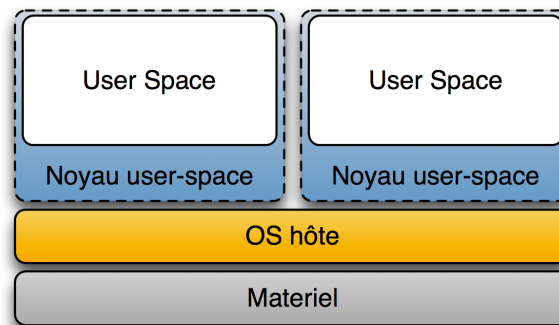


Figure 1.3: Architecture Noyau en espace utilisateur

**Source : référence [27]**

**Exemples :** User Mode Linux, Cooperative Linux, Adeos, L4Linux.

### 1.1.2.3 Virtualisation complète

Ici, un OS de base exécute des logiciels parmi lesquels la VMM, qui à son tour exécute des VMs dans l'espace utilisateur.

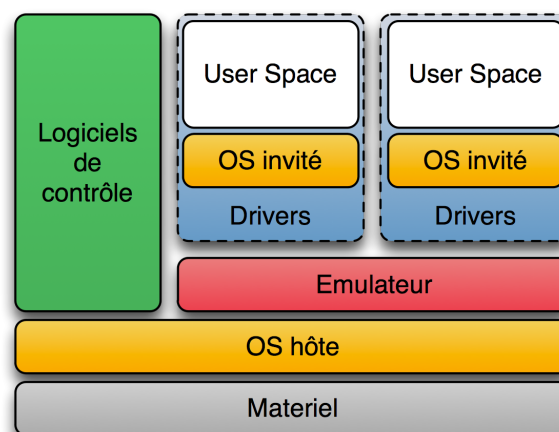


Figure 1.4: Architecture Virtualisation complète

**Source : référence [27]**



La VMM émule le matériel pour les OS invités de façon à ce que ces derniers croient communiquer directement avec ledit matériel. L'OS de la VM est non modifié et peut être de n'importe quel type (Linux, Windows, etc.). Cette solution isole bien les OS invités, mais elle a un coût en performance qui peut être très élevé. La figure 1.4 présente l'architecture de cette technique de virtualisation.

**Exemples :** Oracle VM VirtualBox, QEMU, KVM, Microsoft VirtualPC, VMware Workstation.

#### 1.1.2.4 Para-virtualisation

La VMM (ici appelée hyperviseur) remplace l'OS hôte et sert d'intermédiaire pour communiquer avec le matériel. Les OS invités fonctionnent en ayant conscience d'être virtualisés et sont optimisés pour ce fait. L'OS hôte est lui-même considéré comme une VM (privilégiée) et est utilisé par la VMM pour assurer certaines tâches. C'est la méthode de virtualisation la plus performante, mais elle a pour contrainte d'exiger la modification des OS des VMs. La figure 1.5 présente l'architecture de cette technique de virtualisation.

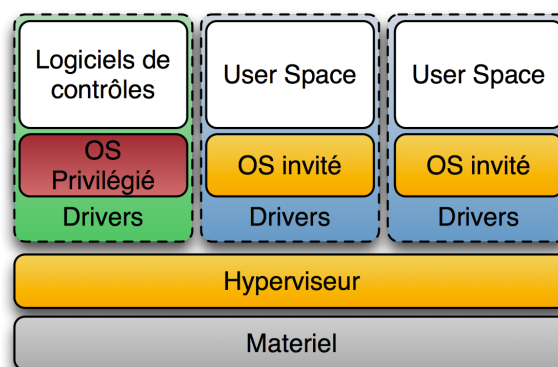


Figure 1.5: Architecture Para-virtualisation  
**Source : référence [27]**

**Exemples :** Xen, VMware vSphere, Microsoft Hyper-V Server, Parallels Server Bare Metal.







### 1.1.2.5 Virtualisation assistée par le matériel

C'est une sorte de para-virtualisation sans intervention de l'OS hôte et dans laquelle les OS des VMs ne sont plus modifiés. Ici, le matériel est au courant de la virtualisation et se charge, par exemple, de virtualiser les accès mémoire ou de protéger le processeur physique des accès les plus bas niveaux. La collaboration du matériel permet de simplifier la virtualisation logicielle et de réduire la dégradation de performances. En outre, ici les VMs portent le nom de Hardware Virtual Machine (HVM).

**Exemples :** AMD-V (assistance à la virtualisation de AMD) et Intel VT (assistance à la virtualisation de Intel).

### 1.1.3 Systèmes de virtualisation

Le tableau suivant (1.1) récapitule certaines VMMs en fonction de leur type de virtualisation :

Tableau 1.1: Systèmes de virtualisation en fonction du type de virtualisation

VMM	Technique de virtualisation
<b>OpenVZ</b>	Niveau OS
<b>Linux-VServer</b>	Niveau OS
<b>L4Linux</b>	Noyau en espace utilisateur
<b>Adeos</b>	Noyau en espace utilisateur
<b>Oracle VM Virtualbox</b>	Virtualisation complète
<b>QEMU</b>	Virtualisation complète
<b>VMWare vSphere</b>	Para-virtualisation
<b>Xen</b>	Para-virtualisation

Le système que nous utilisons dans le cadre de ce travail est **XEN**. *XEN* est un système de virtualisation assez connu [9] et utilisé par Amazon pour virtualiser ses datacenters[7]. Il s'appuie sur un hyperviseur qui s'exécute sur le matériel et une machine virtuelle particulière appelée **dom0** : il s'agit de l'OS de la machine hôte. Les services de l'OS hôte ne sont pas inclus dans l'hyperviseur afin de le maintenir aussi léger que possible. Les autres VMs ici sont appelées **domU** (voir figure 1.6).

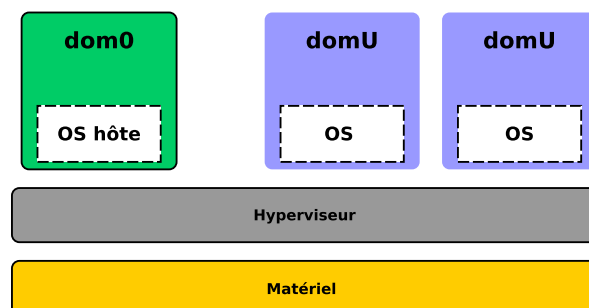


Figure 1.6: Architecture Virtualisation avec hyperviseur XEN

### 1.1.4 Avantages de la virtualisation

Selon le dernier rapport de l'entreprise de conseil et de recherche Gartner Inc. au sujet de la virtualisation, le niveau de pénétration de la virtualisation est assez élevé, avec plusieurs organisations ayant un taux de virtualisation de leurs serveurs excédant les 75% [16]. Cette utilisation de plus en plus importante des techniques de virtualisation dans les DCs doit son mérite aux nombreux avantages qu'elles offrent. Ceux-ci sont relatifs à :

- ☛ **La réduction des coûts** : le fait que plusieurs VMs peuvent s'exécuter sur une même machine hôte entraîne une réduction du nombre de serveurs physiques et donc une réduction des coûts liés à l'acquisition, l'exploitation et l'expansion de l'infrastructure.
- ☛ **L'isolation et l'amélioration de la sécurité** : la virtualisation offre la possibilité de séparer les différentes tâches d'un serveur physique entre plusieurs VMs, ce qui permet donc de les isoler les uns des autres et donc d'accroître le niveau de sécurité.
- ☛ **La minimisation des interruptions et l'amélioration de la disponibilité des services** : les solutions de virtualisation rendent possible la migration de machines virtuelles d'un serveur physique à un autre [5, 12]. Cette fonctionnalité est un élément important qui améliore la disponibilité des services.



- ☛ **L'optimisation des performances** : un autre avantage de la migration de machines virtuelles est qu'elle rend possible de distribuer les charges de travail entre les différents serveurs physiques [5, 12]. Ceci permet donc une meilleure gestion des ressources et donc une optimisation des performances.
- ☛ **La simplification des opérations de sauvegarde** : qui améliore la tolérance aux pannes.

## 1.2 Virtualisation de la mémoire

Bien que la virtualisation présente de nombreux avantages, les coûts qu'elle engendre (surcharge des processeurs, etc.) ne sont pas négligeables. L'une des sources majeures de ces coûts de performance dans les environnements virtualisés c'est la virtualisation de la mémoire, à travers la traduction d'adresses [15, 6]. En effet, les adresses des processus qui s'exécutent dans les VMs ne sont pas des adresses de la mémoire *réelle* (dans le système hôte), mais des adresses virtuelles, qui doivent être traduites en adresses physiques dans la mémoire centrale.

### 1.2.1 Environnement natif (sans virtualisation)

Dans un environnement natif, les adresses utilisées par un processus (ou application) sont des adresses virtuelles (Vaddr, pour *virtual address*). Chaque processus maintient une table de pages qui contient les correspondances entre ces adresses et les adresses physiques correspondantes (voir figure 1.7).

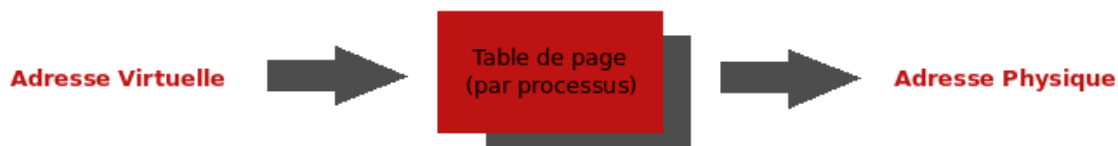


Figure 1.7: Translation adresse virtuelle - adresse physique (environnement natif)

On a donc un seul niveau de parcours de table de pages dans un environnement natif. La figure 1.8 illustre ce parcours.



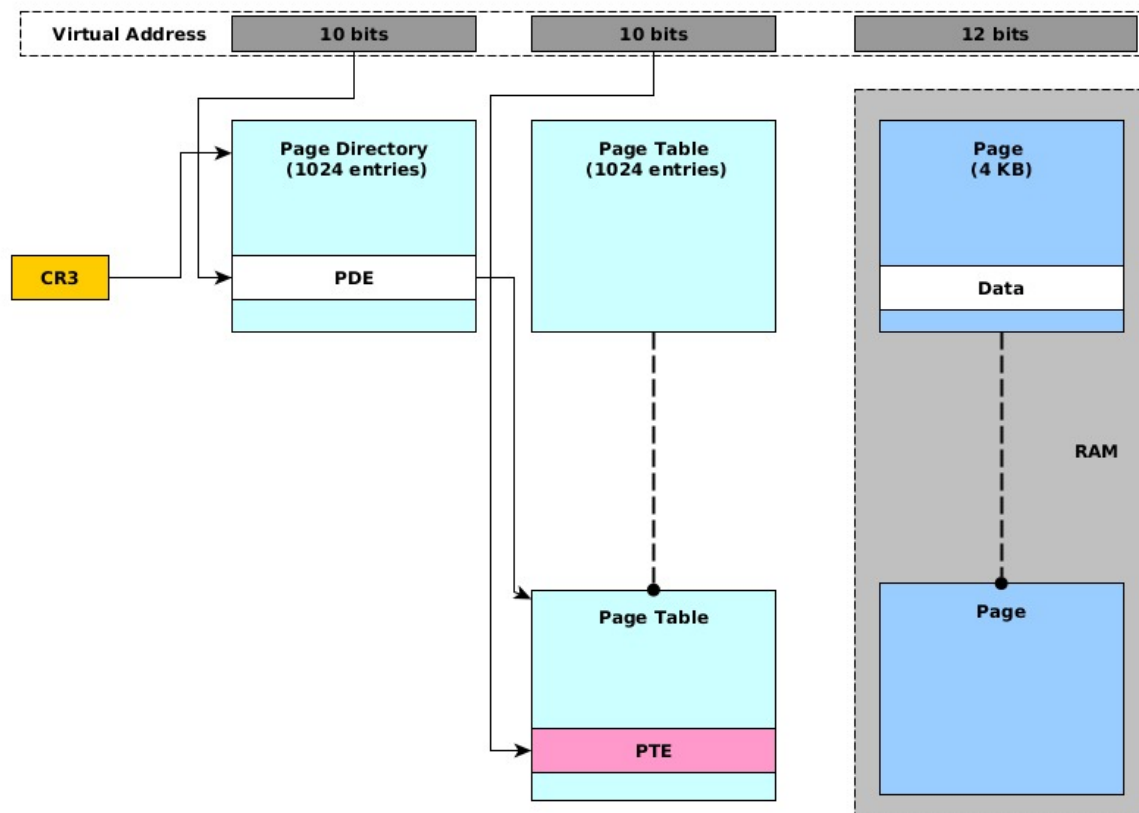


Figure 1.8: Parcours table de page 1D (une dimension)

### 1.2.2 Environnement virtualisé

Dans un environnement virtualisé chaque VM est un OS à part entière, donc les processus qui s'y exécutent maintiennent une table de page (*guest Page Table, gPT*), comme dans un environnement natif, pour effectuer la translation d'adresses virtuelles, appelées ici *guest Virtual Address (gVA)*, en adresses physiques ***guest Physical Address (gPA)***. Ici, deux approches ou mécanismes de translation sont utilisés : ***shadow paging*** et ***nested paging***.

Avant d'aborder ces notions, nous allons parler en quelques mots de la TLB (*Translation Lookaside Buffer*).



### 1.2.2.1 Translation Lookaside Buffer (TLB)

La TLB est une mémoire *cache* utilisée pour réduire le temps d'accès à la mémoire centrale [23, 8]. Elle enregistre les données récentes des tables de pages, à savoir les translations d'adresses récentes ***Vaddr***  $\rightarrow$  ***hPA***.

Il existe une entrée de la table de pages pour l'intégralité de l'espace d'adressage de chaque programme. Mais étant donné que la table de pages est en mémoire centrale, cela ralentirait le processeur d'aller chercher la correspondance de chaque adresse virtuelle en mémoire centrale. Ainsi la TLB contient les données les plus récemment chargées à partir des tables de pages à l'intérieur du processeur lui-même. Si l'adresse recherchée est trouvée dans la TLB, le processeur peut directement aller à l'emplacement mémoire correspondant. Dans le cas contraire on parle de ***TLB miss***. Le processeur doit donc aller en mémoire trouver la table de pages et charger dans la TLB l'entrée de la table correspondant à l'adresse recherchée. La figure 1.9 présente le fonctionnement global de la TLB.

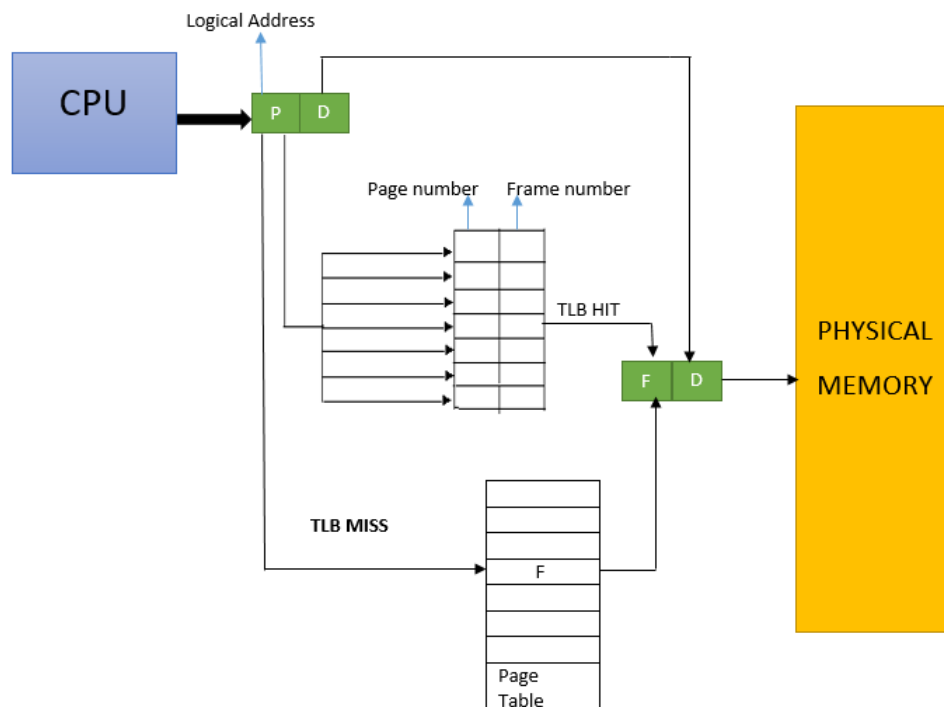


Figure 1.9: Fonctionnement global de la TLB

**Source : référence[29]**





### 1.2.2.2 Shadow paging

#### (i) Description

Dans cette approche, l'hyperviseur maintient une table de pages dont les entrées sont directement les correspondances **gVA**  $\rightarrow$  **hPA**. Cette table porte le nom de **shadow page table**.

A chaque tentative de la VM de mettre à jour sa table de page, l'hyperviseur capture les translations **gVA**  $\rightarrow$  **gPA** et **gPA**  $\rightarrow$  **hPA** pour construire le shadow page table. En cas de TLB miss, le processeur n'a plus qu'à effectuer un parcours 1D (une dimension) de cette table de page pour trouver l'adresse virtuelle correspondante. La figure 1.10 résume ce mécanisme.

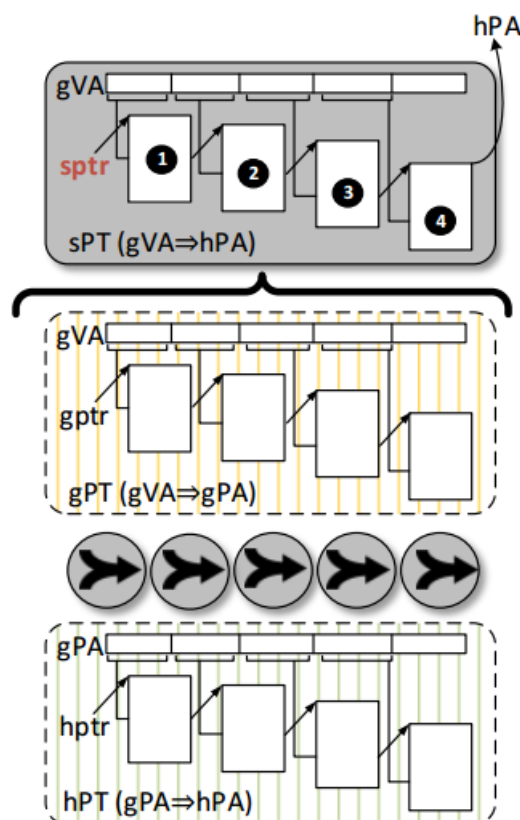


Figure 1.10: Mécanisme du shadow paging  
Source : (ISCA'16 [15])

#### (ii) Avantages et Inconvénients

L'approche *shadow paging*, présente les avantages suivants :



- Uniquement quatre accès mémoire requis en cas de TLB miss.
- Pas besoin d'un support matériel supplémentaire pour le parcours de la table de pages.

À côté de ces avantages, les points négatifs suivants sont à noter :

- Toute mise à jour de la table de pages génère une exception qui sera gérée par l'hyperviseur.
- Tout changement de contexte entraîne des détournements coûteux dans l'hyperviseur.

### 1.2.2.3 Nested paging

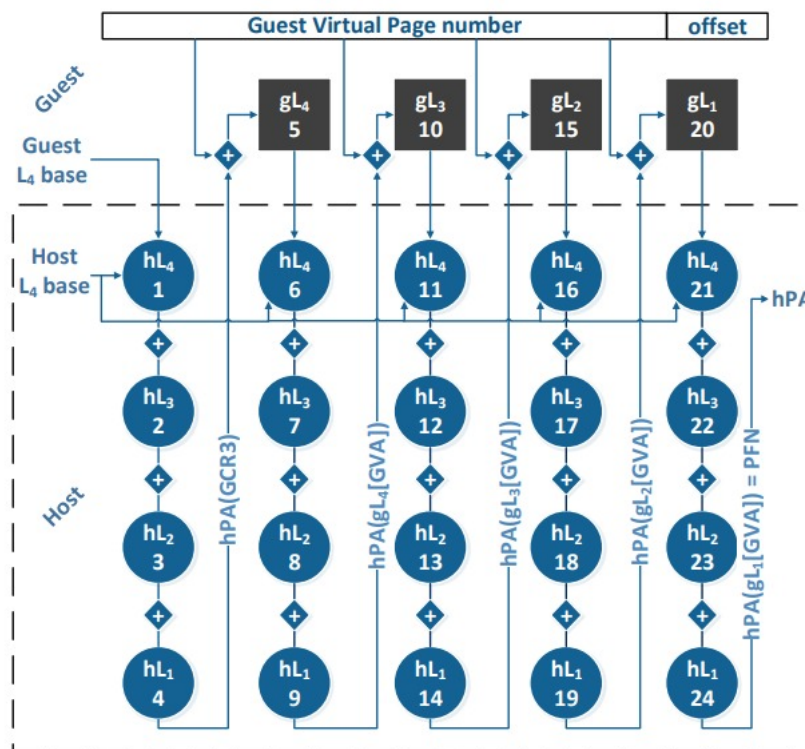


Figure 1.11: Mécanisme du nested paging

Source : (référence [6])

#### (i) Description

Cette approche est utilisée dans le cadre de la virtualisation assistée par le



matériel. Ici la table de pages est imbriquée (*nested*), car la translation d'adresses se fait en tenant compte de deux niveaux de table de pages : une dans la VM (gPT), et l'autre dans l'hyperviseur. De plus, le matériel (processeur) est au courant de cette deuxième table de page dans l'hyperviseur : on parle de virtualisation complète de la mémoire.

Certains fabricants de processeurs ont développé des fonctionnalités pour prendre en compte cette approche de virtualisation de la mémoire :

- AMD avec le ***Nested Page Table (NPT)***
- Intel avec l'***Extended Page Table (EPT)***

La figure 1.11 décrit le parcours 2D (2 dimensions) d'une table de page imbriquée (*nested page table*).

## (ii) Avantages et Inconvénients

Avec cette approche, la mise à jour des tables de pages se fait rapidement et de façon directe, sans aucun détournement dans l'hyperviseur.

Le principal inconvénient est le nombre d'accès mémoire nécessaires en cas de TLB miss (24 au total, voir figure 1.11).

## 1.3 Page Modification Logging (PML)

### 1.3.1 Intel VT-x

Pour mieux comprendre ce qu'est le PML, il est nécessaire de connaître certaines notions relatives aux supports matériels des processeurs Intel pour la virtualisation.

Dans le cadre de la virtualisation assistée par le matériel (sous-section 1.1.2.5), la virtualisation s'étend à la mémoire et au processeur. Dans la sous-section 1.2.2.3, nous avons fait allusion à l'EPT qui est la technologie introduite par Intel pour la virtualisation de la mémoire. En ce qui concerne la virtualisation du CPU (processeur), Intel a introduit les VT-x (Intel Virtual Technology Extensions).







Les extensions VT-x dupliquent entièrement l'état architectural visible du processeur et introduisent un nouveau mode d'exécution, le **mode root**. Ainsi, l'hyperviseur et l'OS hôte s'exécutent en mode root (avec tous les droits et privilèges), et les autres VMs en *mode no-root* (avec des accès restreints).

Avec les VT-x, chaque VM a une structure de données en mémoire centrale, gérée par la VMM et appelée **Virtual Machine Control Structure (VMCS)**. Cette structure de contrôle sert à sauvegarder l'état de la VM. Lorsque l'état de la VM est chargée, cette dernière s'exécute en mode non-root jusqu'à ce qu'elle génère une exception qui doit être gérée par l'hyperviseur : cette transition (la main passe de la VM à l'hyperviseur qui doit gérer l'exception) du mode non-root au mode root est appelée **VMExit**.

### 1.3.2 Description du PML

Le PML est utilisé dans le cadre de la virtualisation. C'est une amélioration de la technologie *Intel Virtualization Technology (Intel VT)*. Il étend les capacités des VMM qui utilisent le mécanisme d'EPT, en leur donnant la possibilité de traquer les pages mémoire modifiées par les VMs (*guest-physical pages*) durant leur exécution [17].

Intel a inclu dans ses processeurs récents des bits Accessed and Dirty (A/D) <sup>1</sup> pour l'EPT [13]. Etant donné que ces bits sont définis ou mis à jour sans invoquer la VMM, elle n'a aucun moyen d'accumuler des statistiques de *working set* (pages utilisées, modifiées, etc.) pendant l'exécution de la VM. Pour obtenir de telles statistiques, la VMM doit scanner à tout moment l'EPT pour collecter les informations sur les bits *accessed* et *dirty*, ou alors marquer les pages comme *non présentes* ou *en lecture seule*, de sorte que si un processus tente d'accéder à une page ou de la modifier, cela génère une exception que la VMM pourra capturer. De telles opérations imposent des coûts de latence et de performance qui sont inacceptables dans certaines circonstances.

Le PML s'appuie sur la prise en charge par le processeur des bits A/D de l'EPT (sous-section 1.2.2.3 : virtualisation complète de la mémoire). En effet, il étend le procédé qui

---

<sup>1</sup>Lorsqu'un processus accède à une page mémoire, pendant la translation d'adresse lors du parcours de l'EPT, le bit *accessed* correspondant est mis à 1. De même le bit *dirty* est mis à 1 pour signifier que la page a été modifiée.



se produit lorsque les bits *dirty* sont mis à jour (quand une page mémoire est modifiée). Lorsqu'une écriture fait passer un bit *dirty* de 0 à 1, le processeur enregistre l'adresse physique (gPA) de la page mémoire<sup>2</sup> à l'origine de cet accès dans un emplacement mémoire sous le nom de *page modification log*. Lorsque cette page de logs est pleine (512 entrées), le processeur génère un *VMExit*. Pendant ce *VMExit*, aucune adresse supplémentaire n'est enregistrée<sup>3</sup>, et le processeur met effectivement à jour les bits *dirty* des adresses correspondantes dans une structure de données associée à chaque VM et appelée *bitmap* (voir paragraphe 3.1.1).

L'introduction du PML a nécessité des changements spécifiques dans les VMCS :

- ***enable PML*** : c'est le bit 17 du registre de contrôle spécifique «*secondary processor-based VMexecution control*». Il doit être mis à 1 pour activer le mécanisme du PML (lorsqu'il est supporté par le processeur).
- ***Page modification log*** : c'est l'espace mémoire dans lequel sont enregistrées les gPA. Cette page comprend 512 entrées de 64 bits.
- ***PML address*** : c'est un nouveau champ de 64 bits dans le registre de contrôle spécifique «*secondary processor-based VM execution control*». Il représente l'adresse physique<sup>4</sup> du *page modification log*.
- ***PML index*** : c'est également un nouveau champ, de 16 bits. Il représente l'index de la prochaine entrée dans le *page modification log*. Etant donné que le *page modification log* n'enregistre que 512 entrées, la valeur de cet index est compris entre 0 et 511.

Dans ce chapitre, il a été question tout d'abord d'introduire des généralités sur la virtualisation (les types et techniques de virtualisation), puis de parler de la virtuali-

---

<sup>2</sup>Il s'agit des *guest-physical addresses*, i.e. des pages mémoire des processus s'exécutant dans la VM.

<sup>3</sup>Car pendant un *VMExit*, la VM est en pause étant donné que l'hyperviseur prend la main.

<sup>4</sup>Page mémoire de 4-KBytes



sation de la mémoire (en évoquant les notions de TLB, *shadow* et *nested* paging) et enfin d'introduire la notion de PML.

## ETAT DE L'ART SUR L'ESTIMATION DU WORKING SET



*Ce chapitre fait le tour des différents travaux qui proposent des solutions au problème d'estimation du WSS.*



## 2.1 Enoncé du problème

Comme mentionné à l'introduction, la mémoire est la ressource critique en terme de consolidation de VMs. Pour pallier ce problème, la solution la plus communément utilisée consiste à gérer la mémoire de la même façon que le processeur, i.e. d'effectuer une allocation de mémoire à la demande.

### 2.1.1 Allocation de mémoire à la demande

Soit une VM dont la mémoire réservée a une capacité  $m_r$  (représentant le SLA<sup>1</sup>, Service Level Agreement, que le fournisseur ou l'hébergeur doit respecter), mais dont la mémoire activement utilisée est  $m_u$  ( $m_u \leq m_r$ ), l'approche d'allocation à la demande n'allouera à la VM qu'une quantité  $m_u$  de mémoire (au lieu de  $m_r$  comme avec une allocation statique) :  $m_u$  est appelée WSS de la VM [24]. Cette approche nécessite d'implémenter une boucle qui fonctionne comme suit :

- (i) A chaque période  $T$  définie, l'activité de chaque VM est collectée périodiquement et sert d'*input* à l'algorithme d'estimation du WSS;
- (ii) Une fois l'estimation faite (notée  $wss_{est}$ ), la mémoire allouée à la VM est ajustée à l'estimation faite  $wss_{est}$ . On observe la VM pendant un certain moment puis on recommence au (i).

Ceci fait naître deux principales interrogations :

- ☛ **(Q1)** Comment observer la VM et collecter les informations sur son activité sachant que celle-ci est une «**boîte noire**» pour l'hébergeur ?
- ☛ **(Q2)** Une fois qu'on a répondu à (Q1), comment estimer le WSS de la VM à partir des données collectées ?

---

<sup>1</sup>En français entente du niveau de service, c'est un document qui définit la qualité de service attendue et le niveau de service que souhaite obtenir un client de son fournisseur.



### 2.1.1.1 Répondre à Q1 : Comment collecter les informations sur l'activité mémoire de la VM ?

Répondre à (Q1) soulève deux défis :

- Le premier est relatif à la méthode à utiliser pour la collecte des données liées à l'activité mémoire de la VM. Une telle méthode peut être soit **active** soit **passive**. Dans le premier cas, la méthode aura pour conséquence de modifier le cours d'exécution de la VM, ce qui n'est pas le cas pour une méthode passive.

Une méthode active peut impacter sur les performances de la machine virtuelle. Par exemple, une façon triviale de traquer tous les accès mémoire est d'invalidiser (les marquer comme étant *non présentes*) toutes les pages du *shadow page table* (sous-section 1.2.2.2) de la VM. Ainsi tous les accès subséquents produiront des défauts de page qui seront capturés par l'hyperviseur. Cette méthode peut en effet être catastrophique pour les performances de la VM à cause des coûts imputés par un défaut de page.

- Le deuxième défi se rapporte au niveau d'implémentation de la méthode utilisée. En effet, une telle méthode peut être implémentée à trois endroits : soit exclusivement dans l'hyperviseur (ou dans le dom0), soit exclusivement à l'intérieur de la VM, soit réparti à travers les deux. Dans les deux derniers cas, la méthode est dite **intrusive** car la nature *boîte noire de la VM* sera altérée. Dans cette circonstance, l'implémentation de la méthode nécessiterait l'accord du client.

### 2.1.1.2 Répondre à Q2 : Comment estimer le WSS à partir des données collectées ?

Concernant (Q2), les principaux défis à relever sont : la **précision** de l'estimation et le coût lié à l'algorithme utilisé. En effet, étant donné qu'après estimation faite du WSS la mémoire allouée à la VM est réajustée, une estimation erronée pourrait soit impacter sur les performances de la machine, soit causer un gaspillage encore plus conséquent des ressources.



### 2.1.2 Les critères d'estimation du WSS

Les questions soulevées précédemment mettent l'accent sur les métriques à prendre en compte lorsqu'on définit un algorithme d'estimation pour le WSS :

- ☛ **Intrusive.** Une bonne méthode d'estimation ne devrait pas être intrusive, i.e. nécessiter la modification de la VM.
- ☛ **Active.** Une méthode active altère le flow d'exécution de la machine, ce qui n'est pas acceptable pour le client.
- ☛ **Précise.** L'estimation faite doit être précise pour éviter un réajustement erroné de la mémoire allouée à la VM.
- ☛ **Surcharge de la VM.** Une bonne méthode d'estimation ne doit pas engendrer des dégradations de performance de la VM. Ceci peut être la conséquence directe d'une méthode intrusive ou active.
- ☛ **Surcharge de l'hyperviseur.** Une surcharge importante peut saturer l'hyperviseur ou le dom0, ce qui pourrait conduire à des dégradations de performance des VMs.

## 2.2 Les techniques existantes

Dans cette section nous abordons les principales techniques proposées par des chercheurs. Pour chacune d'elles :

- Nous présentons d'abord une brève description en montrant comment la technique répond aux questions Q1 & Q2
- Ensuite nous expliquons sommairement la méthode d'implémentation dans XEN (qui est le système de virtualisation que nous utilisons)
- Et enfin nous présentons les forces et faiblesses de la solution





## 2.2.1 Self-ballooning

### 2.2.1.1 Description

Le self-ballooning [1] repose essentiellement sur la VM, particulièrement sur son OS.

**Réponse à Q1.** Cette technique considère que le WSS de la VM est donné par les statistiques du noyau, à l'occurrence le *Committed\_As*[2, 3]<sup>2</sup> (*cat /proc/mem/info*). Donc c'est l'OS lui même qui observe la VM et gère les appels d'allocation mémoire (*malloc*, *etc.*), et calcule la mémoire totale allouée à tous les processus.

**Réponse à Q2.** L'OS décrémente le *Committed\_As* chaque fois que des pages mémoire allouées sont libérées. Par exemple, considérons le processus qui exécute le programme *C* à la figure 2.1 : après l'exécution de la ligne 2, la valeur du *Committed\_As* est incrémentée de 2GB, même si uniquement 1 octet est effectivement utilisé.

```
1 void main(void){
2     char* tab=(char*)malloc(2*1024*1024*1024);
3     do{
4         tab[1]=getchar();
5     }while(tab[1]!='a');
6     free(tab);
7 }
```

Figure 2.1: Augmentation de la valeur du *Committed\_As* avec l'instruction *malloc* même si ne correspondant pas à la valeur de la mémoire physique utilisée

En somme, le *Committed\_As* correspond au nombre total de pages mémoire virtuelles allouées par un processus, même si elles ne correspondent pas forcément à des pages physiques en mémoire centrale.

### 2.2.1.2 Méthode d'implémentation

Aucun effort d'implémentation n'est requis pour cette solution car c'est la technique utilisée par défaut dans XEN. En effet, XEN comme la plupart des hyperviseurs aujourd'hui,

<sup>2</sup>Quantité de mémoire, en kilooctets, nécessaire à la VM pour exécuter sa charge de travail. Elle représente la valeur du scénario le plus défavorable, incluant les défauts de page.





implémente le *memory ballooning*[24, 31]. C'est une technique de gestion mémoire qui permet à l'hyperviseur de réclamer de la mémoire aux VMs et de la réallouer à d'autres. Dans cette technique, l'hyperviseur équipe chaque VM d'un *balloon driver* qui peut être *gonflé* ou *dégonflé* dans le but d'augmenter ou de diminuer la mémoire allouée à la VM. Ainsi, en fonction de la valeur du *Committed\_As*, l'hyperviseur va dynamiquement envoyer des instructions au *balloon driver* de la VM qui va réajuster la mémoire de cette dernière.

### 2.2.1.3 Caractéristiques et limites

Comme présenté plus haut, cette méthode repose entièrement sur la VM. En outre, c'est le *balloon driver* situé dans la VM qui se charge d'augmenter ou de diminuer le *cache* de la machine, ce qui rend la solution très **intrusive**.

D'autre part, l'heuristique utilisée pour l'estimation du WSS, à savoir la valeur du *Committed\_As*, n'est pas précise car elle ne prend pas en compte la mémoire cache. De ce fait, la mémoire estimée est la plupart du temps plus grande que celle activement utilisée par la VM, ce qui peut conduire à un gaspillage encore plus considérable des ressources.

## 2.2.2 Zballoond

### 2.2.2.1 Description

*Zballoond* [11] repose sur l'observation suivante : lorsque la taille de la mémoire allouée à la VM est supérieure à son WSS, le nombre de *swap* est presque nul.

**Réponse à Q1.** L'idée basique derrière *Zballoond* est de diminuer graduellement la mémoire de la VM jusqu'à ce que le nombre de *swaps* commence à augmenter (devenir différent de zéro).

**Réponse à Q2.** Avec ce procédé, le WSS de la VM est donc considéré comme la plus petite taille mémoire qui maintient le nombre de *swaps* à zéro.





### 2.2.2.2 Méthode d'implémentation

*Zballoond* est implémenté à l'intérieur de la VM comme un module du noyau. L'algorithme d'implémentation boucle entre les étapes suivantes :

- (i) La mémoire de la VM est initialisée à sa valeur du *Committed\_As*
- (ii) A chaque période (1 seconde par exemple), la mémoire est décrémentée par pourcentage du *Committed\_As* (par exemple 5%)
- (iii) Dès que la valeur du *Committed\_As* change, *Zballoond* considère que le WSS de la VM a également changé, et l'algorithme retourne à l'étape 1

### 2.2.2.3 Caractéristiques et limites

Comme la méthode précédente, *Zballoond* est entièrement implémentée dans l'OS de la VM ce qui la rend totalement **intrusive**.

De plus, c'est une technique très **active** en ce sens qu'elle force l'OS de la VM à réclamer des pages mémoire (à chaque période de l'étape de l'algorithme). Ainsi, les coûts imputés par cette méthode dépendent de la période d'observation et de la pression exercée sur la VM (quantité de mémoire réclamée).

## 2.2.3 VMware

### 2.2.3.1 Description

*VMware* [31] est une amélioration de la technique naïve d'allocation à la demande. Au lieu d'invalider toutes les pages, la technique s'appuie sur une approche d'échantillonnage. Soit  $m_{act}$  la taille actuelle de la mémoire de la VM :

**Réponse à Q1.** L'hyperviseur choisit périodiquement et aléatoirement  $n$  pages dans la mémoire de la VM et les rend invalides (en les marquant *non présentes* par exemple). Ainsi les prochains accès à ces pages généreront des exceptions qui seront capturées par l'hyperviseur.





**Réponse à Q2.** Une fois ces exceptions générées, l'hyperviseur compte le nombre  $f$  de pages, parmi les pages précédemment sélectionnées dans l'échantillon, qui ont été sujettes à des défauts de page (pages marquées non présentes auxquelles la VM a tenté d'accéder). De cette façon, le WSS de la VM est estimé par la formule  $\frac{f}{n} * m_{act}$ .

### 2.2.3.2 Méthode d'implémentation

Cette technique peut être implémentée de deux manières dépendamment de la façon dont les pages sont invalidées : soit en effaçant<sup>3</sup> le bit de présence, soit en effaçant le bit *accessed*.

Entre le moment où les pages sont invalidées et celui où l'estimation du WSS est faite, l'hyperviseur va compter soit :

- Le nombre de défauts de pages survenus, dans le cas où le bit de présence a été mis à 0.
- Le nombre de pages auxquelles la VM a accédées (parmi les pages de l'échantillon), dans le cas où le bit *accessed* a été mis à 0. Dans ce second cas de figure, l'hyperviseur n'a pas à capturer une exception car le bit *accessed* est automatiquement mis à 1 par le matériel lorsqu'un processus accède à une page.

### 2.2.3.3 Caractéristiques et limites

Cette technique est **non intrusive** car entièrement implémentée dans l'hyperviseur/dom0. Toutefois, elle présente deux inconvénients majeurs :

- Le fait d'invalider les pages mémoire de la VM modifie son cours d'exécution, ce qui peut provoquer des dégradations de performances en fonction de la méthode d'implémentation : dans le cas où l'implémentation se fait en effaçant le bit de présence des pages, les dégradations sont plus importantes (dus aux coûts de résolution d'un défaut de page). Toutefois, dans le cas c'est le bit *accessed* qui est

---

<sup>3</sup>Mettre le bit correspondant à 0



mis à 0, la précision de l'estimation peut être biaisée si l'hyperviseur/dom0 exécute d'autres programmes susceptibles de remettre ce bit à 0.

- *VMware* est incapable d'estimer un WSS supérieur à la mémoire actuellement allouée à la VM. En effet, dans le meilleur des cas l'algorithme déterminera que toutes les pages de la machine ont été accédées et estimera ainsi le WSS à la taille actuelle de la mémoire.

## 2.2.4 Geiger

### 2.2.4.1 Description

*Geiger* [19] répond aux questions 1&2 tel qu'il suit.

**Réponse à Q1.** Pour observer l'activité mémoire de la VM, la technique consiste à surveiller les évictions et mise à jour éventuelles du cache de la VM vers la partition de swap.

**Réponse à Q2.** *Geiger* repose sur une technique appelée *buffer fantôme* [25]. Ce dernier est un buffer imaginaire qui étend la mémoire physique de la VM ( $m_{act}$ ). La taille de ce buffer représente la quantité de mémoire supplémentaire qui doit permettre d'éviter à la VM d'effectuer des *swap out* de pages mémoire. Ainsi l'estimation du WSS de la VM se fait à travers la formule :  $WSS = m_{act} + m_{fant}$  si  $m_{fant} > 0$ .

### 2.2.4.2 Méthode d'implémentation

Le challenge pour implémenter cette technique est de pouvoir isoler le trafic du swap des autres requêtes d'entrée/sortie, dans le but de forcer la VM à utiliser un autre pilote comme partition de swap. Ce pilote est spécialement configuré pour l'implémentation de la technique, de la façon suivante : lorsqu'une page est éjectée de la mémoire de la VM, une référence à cette page est ajoutée à une file dans le pilote situé dans le dom0. Plus tard, lorsqu'une page est lue depuis la partition de swap, *Geiger* retire la référence à cette page de la file et calcule la distance  $D$  de la page à la tête de file. C'est cette





distance  $D$  qui représente le nombre de pages mémoire supplémentaires, nécessaire à l'OS de la VM pour éviter des *swap out*.

### 2.2.4.3 Caractéristiques et limites

Tout comme *VMware*, *Geiger* est totalement transparente du point de vue de la VM, et ne nécessite donc pas de permission de la part du client. Toutefois ce caractère non intrusif de la technique pose un sérieux problème. En effet, *Geiger* n'est capable d'estimer le WSS que si la taille de la *mémoire fantôme* est strictement positive, i.e. si la VM est dans un état de *swap*. *Geiger* devient donc inefficace si le WSS de la machine est inférieure à la mémoire qui lui est actuellement allouée.

## 2.2.5 Hypervisor Exclusive Cache

### 2.2.5.1 Description

*Exclusive Cache* [21] est assez similaire de *Geiger* en ce sens qu'il repose également sur la notion de buffer fantôme pour estimer le WSS.

Ici chaque VM détient une petite quantité de mémoire appelée mémoire directe, le reste étant gérée par l'hyperviseur sous forme de **cache exclusif**. C'est ce cache exclusif qui est considéré dans cette technique comme la mémoire fantôme (dans le cas de *Geiger*). Ainsi, une fois que la VM consomme toute sa mémoire directe, elle swap ses pages non pas vers la partition de swap mais vers l'hyperviseur.

### 2.2.5.2 Méthode d'implémentation

Tout comme *Geiger*, *Exclusive Cache* est implémenté sous forme d'une extension disque. Mais ici à la place d'un pilote de disque externe comme avec *Geiger*, le contenu des pages de la VM peut être sauvegardé dans un buffer situé dans le dom0, qui matérialisera le buffer fantôme.





### 2.2.5.3 Caractéristiques et limites

À l'opposé de *Geiger*, cette technique est plus active dans la mesure où elle oblige la VM à être dans un état de swap, étant donné qu'il y a une partie de sa mémoire qui est conservée par l'hyperviseur.

Quoiqu'il en soit, les impacts de performance sont moindres car l'extension de disque est contournée et les pages mémoire de la VM sont enregistrées dans le dom0.

## 2.2.6 Dynamic MPA ballooning

### 2.2.6.1 Description

Le *Dynamic MPA*<sup>4</sup> *ballooning* [20] étudie la gestion de la mémoire du point de vue de l'ensemble de la machine hôte.

**Réponse à Q1.** *MPA ballooning* introduit un ensemble d'hypercalls à travers lesquels toutes les VMs reportent à l'hyperviseur le nombre de leurs pages mémoire.

**Réponse à Q2.** Sur la base de ces informations, l'algorithme définit trois niveaux possibles de pression de la mémoire :

- Bas : le nombre de pages mémoire de toutes les VMs est inférieur à la mémoire physique totale de l'hôte.
- Moyen : le nombre de pages mémoire de toutes les VMs est égal à la mémoire physique totale de l'hôte.
- Elevé : le nombre de pages mémoire de toutes les VMs est supérieur à la mémoire physique totale de l'hôte.

En fonction de l'état de pression de la mémoire, l'OS adopte une politique de gestion différente :

- Dans le cas d'un niveau de pression bas, cette technique divise la mémoire disponible dans l'hyperviseur en  $(nb_{VMs} + 2)$  parts. Chaque part est assignée à une

---

<sup>4</sup>Dynamic Memory Pressure Aware



VM comme mémoire réservée et les deux autres sont maintenues par l'hyperviseur en cas de demande soudaine. Chaque part est vue comme le cache exclusif dans la méthode *Hypervisor Exclusive Cache*.

- Dans le cas d'un niveau de pression moyen, l'hyperviseur réclame les pages inactives à toutes les VMs et les rééquilibre en  $(nb_VMs + 1)$  parts.
- Enfin dans le cas d'un niveau de pression élevé, la technique éjecte toutes les pages du cache et rééquilibre exclusivement les pages anonymes.

### 2.2.6.2 Caractéristiques et limites


*Dynamic MPA* est très intrusif étant donné qu'il nécessite des hypercalls supplémentaires de l'OS de la VM. C'est une technique qui n'est donc réellement effective que dans le cas d'un datacenter privé où le gestionnaire a un contrôle plus élevé sur les VMs. En outre, ces nouveaux hypercalls exportent des données précises et importantes sur la couche mémoire des VMs, ce qui accroît le risque d'attaques sur les machines.

Il était question dans ce chapitre premièrement de présenter les critères à prendre en compte lors de la définition d'une technique d'estimation du WSS, et ensuite les techniques existantes, en disant pour chacune d'elles si elle répond à ces critères et pourquoi. Le tableau 2.1 suivant récapitule pour chacune des techniques existantes avec, les critères d'estimation :

Tableau 2.1: Tableau comparatif des techniques d'estimation du WSS existantes

Technique	Intrusive	Active	Précise	Surcharge la VM	Surcharge l'hyperviseur
<b>Self-ballooning</b>	Oui	Non	Non	Non	Non
<b>ZBalloond</b>	Oui	Oui	Non	Oui	Non
<b>VMWare</b>	Non	Oui	Non	Non	Oui
<b>Geiger</b>	Non	Non	Non si $WSS < \text{mémoire allouée}$	Non	Non
<b>Exclusive-cache</b>	Non	Oui	Non si le cache est nul	Non	Non

## CONTRIBUTION : TECHNIQUE D'ESTIMATION BASÉE SUR UNE ARCHITECTURE AMÉLIORÉE DU PML

e chapitre présente notre contribution à la résolution du problème ci-dessus présenté. Nous montrons au travers de celle-ci comment nous répondons aux questions Q1 et Q2 énoncées au chapitre précédent.



## 3.1 Contribution sur le plan architectural

### 3.1.1 Design actuel du PML

#### 3.1.1.1 Architecture

L'architecture actuelle du PML se présente telle que sur la figure 3.1 suivante :

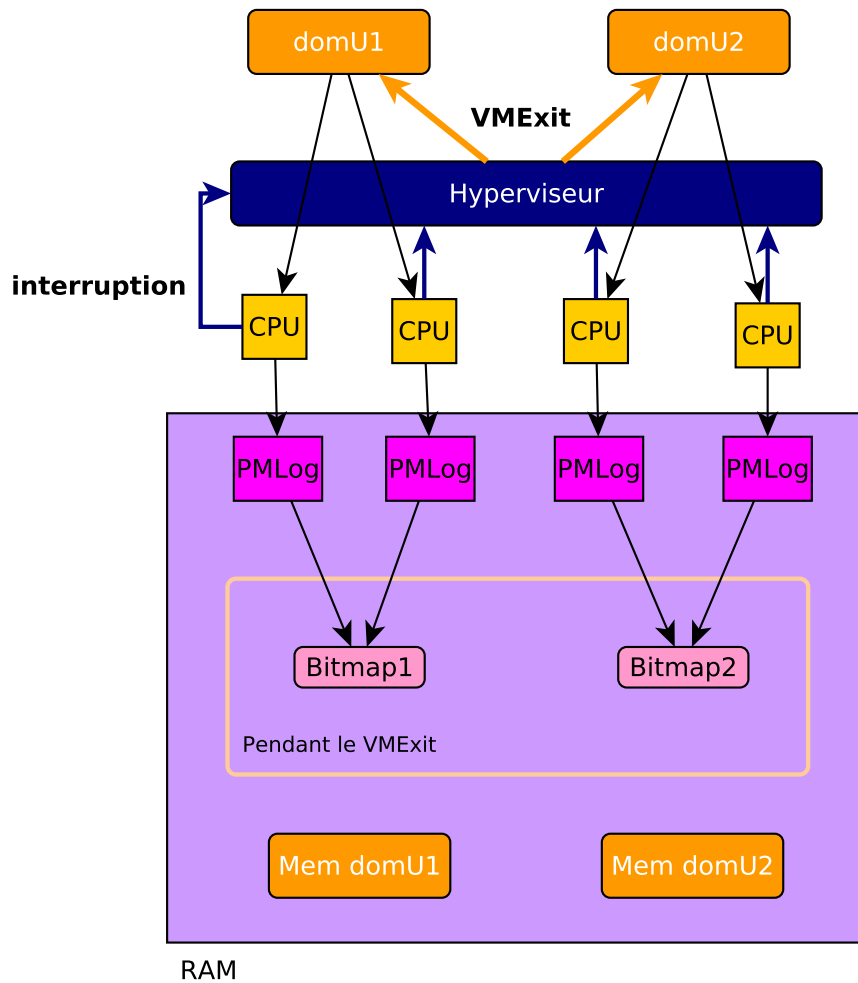


Figure 3.1: Architecture actuelle du PML

Considérons deux VMs *domU1* et *domU2*<sup>1</sup> comme présentées sur la figure précédente. Chacune de ces VMs s'exécute sur des processeurs virtuels, virtual Core Processing Unit (vCPU), rattachés à des processeurs physiques (CPU).

<sup>1</sup>domU pour *domain Unprivileged* en opposition au dom0 qui est l'OS hôte avec tous les droits et accès.



Lorsqu'une VM est créée, si le PML est activé pour celle-ci, alors dans sa structure de contrôle VMCS, l'adresse et l'index du PML sont initialisés (voir 1.3.2). Pendant son exécution :

- (i) Le CPU logue en mémoire centrale, dans le *pml\_log*, les adresses de toute page que la VM modifie. A chaque page modifiée, le CPU ajoute une entrée dans le log.
- (ii) Lorsque le *pml\_log* est plein, i.e. lorsqu'il contient déjà 512 adresses loguées, le CPU sur lequel s'exécute la VM génère une exception *pml buffer full* qui est prise en main par l'hyperviseur.
- (iii) L'hyperviseur capture cette exception et en impute le coup à la VM sous la forme d'un *VMExit*.
- (iv) Pendant ce *VMExit* :
  - Etant donné que la VM ne s'exécute plus, aucune adresse supplémentaire n'est loguée.
  - Le CPU parcourt la bitmap associée à la VM et met à jour le bit correspondant à chaque gPA. Si on veut savoir quelles pages ont été modifiées, il faudrait parcourir cette bitmap et retrouver les adresses correspondant à chaque bit.
  - Et enfin le *pml\_log* est vidé, i.e. la valeur de son index (*PML index*) est remise à 511 <sup>2</sup>.

- (v) Une fois que le *pml\_log* est vidé, la VM résume son exécution et le processus recommence.

Le PML a été introduit pour améliorer :

- La migration des machines virtuelles
- Le checkpointing [4] des machines virtuelles

---

<sup>2</sup>L'index du PML va de 511 à 0.



- Les opérations liées au working set des machines virtuelles (collecte des statistiques de *working set*, estimation du *working set*, etc.)

L'architecture telle que présentée s'est déjà avérée adéquate pour les deux premiers points, mais en ce qui concerne le *working set*, elle présente certaines limites.

### 3.1.1.2 Limites de l'architecture actuelle

Nous présentons ici les principales contraintes auxquelles nous avons fait face avec l'architecture actuelle du PML, et qui l'empêchent d'être efficace pour l'estimation du WSS.

- **Limite 1 : Le VMExit lors de l'interruption *pml buffer full* ne devrait pas être géré par le CPU sur lequel s'exécute la VM dont on estime le WSS**

Lorsque le buffer du *pml\_log* est plein, le CPU génère un *VMExit* qui stoppe l'exécution de la VM qui utilise actuellement le CPU en question, cette VM est celle dont le WSS est calculé. Un changement de contexte est donc effectué pour que l'hyperviseur prenne en main cette interruption. Or les transitions<sup>3</sup> liées aux *VMExits* sont connus pour introduire des coûts significatifs dans la VM [26]. Ainsi accroître le nombre de *VMExits* sur une VM du fait de l'évaluation de son *working set* est inacceptable pour le client car cette opération n'est bénéfique que pour le propriétaire du DC qui veut optimiser l'utilisation de ses ressources et économiser de l'argent. En effet, le client a payé pour une quantité de ressources (ici le temps CPU) qui doit être allouée à sa VM. En générant des *VMExits* supplémentaires, le design actuel du PML réduit le temps CPU réservé par le client. En outre, le temps pris par l'hyperviseur pour capturer cette exception accroît cet impact.

Cette limite n'est un problème ni pour la migration, ni pour le checkpointing car il a été prouvé que réduire le temps CPU durant ces opérations a pour but de les accélérer [22].

---

<sup>3</sup>Passage du mode *non-root* au mode *root*. (Voir sous-section 1.3.1)



- **Limite 2 : La taille du *pml\_log* est très petite (4KB)**

La taille du buffer *pml\_log* a été fixée par Intel à 4KB, pouvant ainsi contenir uniquement 512 gPAs. Cette taille est très petite quand on regarde la plupart des applications de nos jours. La conséquence de ceci est la proportion significative des *VMExits* dus à l'interruption *pml buffer full*, impactant ainsi la VM cible comme présenté à la limite 1. Pour les mêmes raisons que celles présentées à la limite 1, ce problème n'est un obstacle ni pour la migration, ni pour le checkpointing.

- **Limite 3 : Le PML ne devrait pas loguer uniquement les adresses des pages modifiées**

Actuellement, le mécanisme du PML n'enregistre que les pages dont le bit *dirty* passe de 0 à 1, i.e. les pages qui sont modifiées. Or le *working set* d'une VM inclue toutes les pages qu'elle utilise, à la fois les pages modifiées et accédées. Et pour l'instant le PML passe à côté de ces pages auxquelles la VM accède sans les modifier, et donc néglige une bonne partie du *working set*, surtout dans le cas de charges de travail de lecture.

Une fois de plus, ceci n'est pas une préoccupation dans le cadre de la migration ni du checkpointing, car pour ces opérations, juste les pages modifiées sont traquées.

- **Limite 4 : Le PML ne devrait pas loguer les adresses des pages de la table de pages**

En cas de *TLB miss*, le processeur doit aller chercher en mémoire centrale l'adresse physique correspondant au gPA à l'origine du défaut de page. Ceci implique une translation gPA → hPA, et donc un parcours de l'EPT comme présenté sur la figure 1.11 à la sous-section 1.2.2.3. Comme nous le montre cette figure, pendant la translation d'adresses il y a quatre pages intermédiaires qui seront modifiées et donc quatre *guest Physical Addresss* qui seront loguées dans le *pml\_log* en plus de l'adresse initiale à l'origine du défaut de page. Donc pour un gPA à loguer en cas de *TLB miss*, on a quatre adresses superflues qui sont également enregistrées, ce qui remplit inutilement le *pml\_log*, et donc accroît le nombre d'interruptions dues à l'événement *pml buffer full*.

Ceci n'est pas un problème pour les opérations de migration ou de checkpointing, car



pour celles-ci, toute mise à jour d'une table de pages doit être enregistrée, y compris les tables de pages des VMs (par exemple pour la retransmission dans le cas d'une migration en direct).

- **Limite 5 : Le PML devrait tenir compte de la chaleur des pages dans son mécanisme**

En effet, l'estimation du WSS a besoin de connaître la chaleur (le nombre de fois où la page est utilisée) d'une page afin d'être sûr que celle-ci fait bien partie du *working set*. Or pour l'instant, le matériel n'enregistre une page que lorsque son bit *dirty* passe de 0 à 1. Si cette page est de nouveau modifiée dans la suite de l'exécution de la VM, le mécanisme n'en tient plus compte et la page n'est plus loguée étant donné que son bit *dirty* est déjà à 1 et que le matériel ne le remet pas à 0.

Ceci n'est pas un problème pour les autres scénarios car pour la migration on a juste besoin de savoir si une page a été modifiée, le nombre de modifications importe peu.

### 3.1.2 Proposition d'un design amélioré pour l'estimation du WSS

En fonction des limites évoquées plus haut, nous avons conçu un nouveau design architectural dans le but d'améliorer le mécanisme actuel du PML afin qu'il réponde mieux aux besoins d'estimation du WSS. Ainsi l'architecture que nous proposons se décline à travers les étapes suivantes.

#### 3.1.2.1 Redirection des *VMExits* vers le dom0

Cette amélioration permettra de pallier la limite 1. Comme mentionné dans l'énoncé de cette limite, le coût lié à l'évènement *pml buffer full* ne devrait pas être imputé à la VM (sous la forme d'un *VMExit* comme c'est le cas actuellement).

Ainsi, lorsque que le *pml\_log* est plein, nous proposons que le processeur envoie un signal non plus à l'hyperviseur mais au dom0 qui se chargera de traiter cette interruption (Voir figure 3.2).

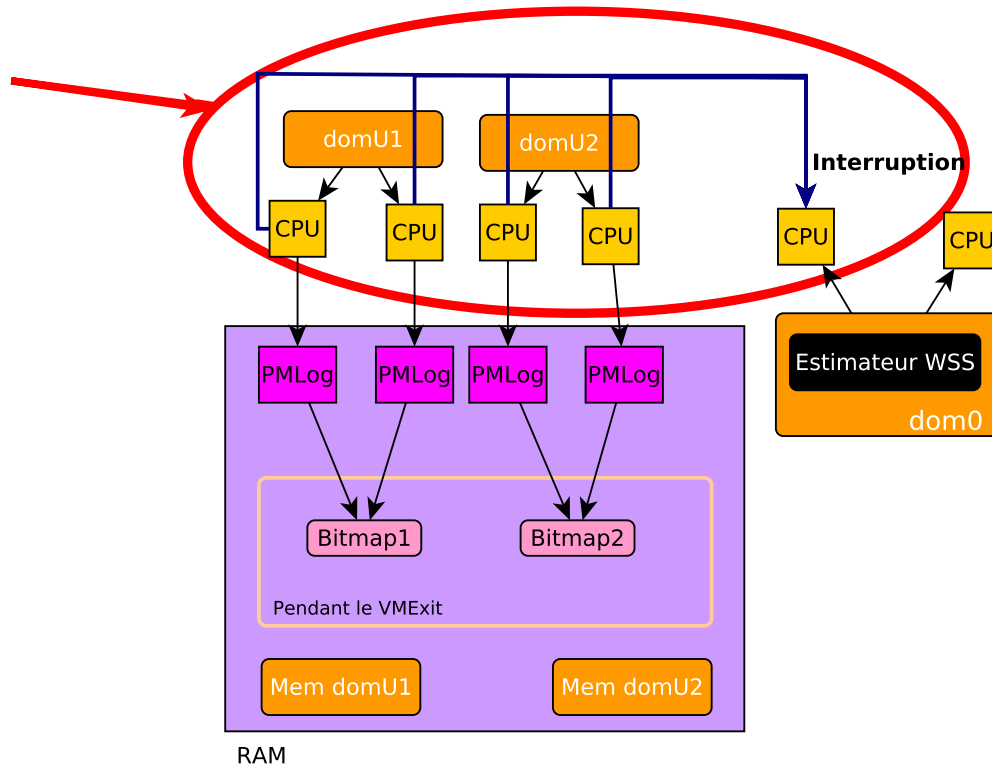


Figure 3.2: Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Redirection des VMExits vers le dom0

### 3.1.2.2 Introduction d'une deuxième page de log (*page modification log*)

La figure 3.3 illustre cette deuxième amélioration qui répond à la limite 2, à savoir la taille insuffisante du *page modification log*.

En outre, cette nouvelle configuration vient en appoint à la première (redirection des *VMExits* vers le dom0). En effet, lorsque le *pml\_log* sera plein, la VM ne va plus arrêter son exécution, or comme nous l'avons mentionné dans la description du mécanisme actuel, lorsque le *pml\_log* est plein aucune adresse supplémentaire ne peut être loguée dans ce dernier. Ainsi, pour éviter de perdre des adresses, nous proposons de rajouter un deuxième *page modification log*, qui pourra continuer d'enregistrer des logs pendant que le premier est vidé. De cette façon, lorsque ce deuxième buffer sera plein, il passera de nouveau la main au premier et ainsi de suite. La VM n'aura donc plus à arrêter son exécution du fait de l'évaluation de son WSS, ceci sans empêcher de collecter les

statistiques nécessaires à l'estimation.

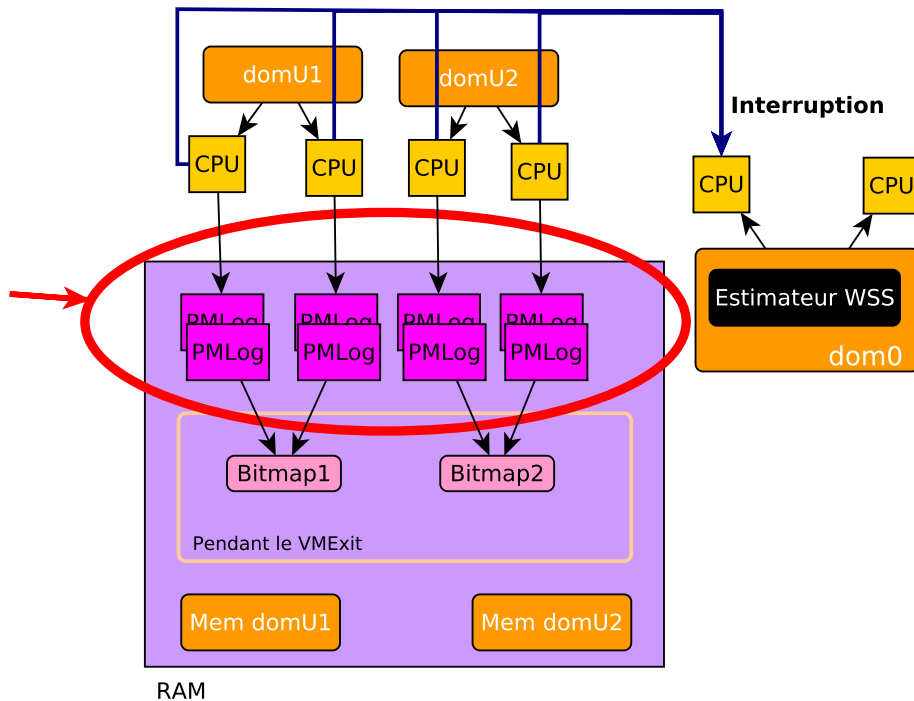


Figure 3.3: Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Introduction d'un deuxième buffer *pml\_log*

### 3.1.2.3 Remettre à 0 le bit *dirty* des pages après avoir vidé la page de logs

Ceci permettra de répondre aux limites 4 & 5. En effet, comme nous l'avons expliqué dans la limite 5, une fois que le bit dirty d'une page passe de 0 à 1, le matériel ne le remet pas à 0. Une solution naïve serait de parcourir la table de pages et remettre le bit *dirty* à 0, mais ceci est une opération très coûteuse. Nous proposons donc une amélioration qui consisterait à loguer par le matériel l'entrée de la table de pages où la page modifiée a été trouvée. En ayant cette information, il sera aisé de remettre à 0 les bits des pages en évitant les pages de la table de pages (ce qui répond à la limite 4). Pour identifier celles-ci, nous proposons d'utiliser un des bits réservés des tables de pages. Ce bit sera positionné à 1 chaque fois qu'il y aura un défaut de page concernant une page de la table de pages.

### 3.1.2.4 Modification de la structure de données *bitmap*

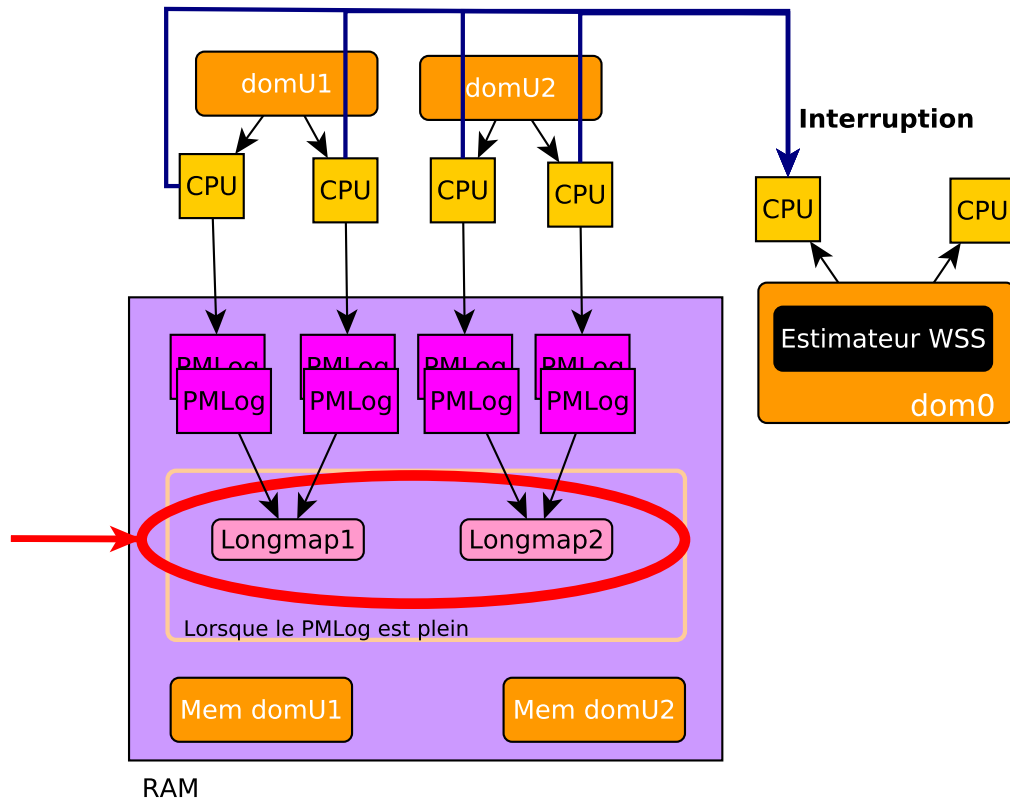


Figure 3.4: Proposition d'une architecture améliorée du PML pour l'estimation du WSS : Modification de la structure de données *bitmap*

Lorsque le *pml\_log* est plein, avant de réinitialiser son index, les informations qu'il contient doivent être consignées. Actuellement ceci est fait au moyen d'une structure de données appelée *bitmap*<sup>4</sup>.

Chaque VM a en mémoire centrale une *bitmap* qui lui est rattachée. Comme son nom y fait référence, elle est constituée de bits représentant chacune une adresse. Lorsqu'une adresse est dans le *pml\_log*, si le bit y correspondant dans la *bitmap* est à 0, il est mis à 1. De cette façon, il nous est difficile d'avoir plus d'informations sur les données enregistrées telles que les adresses en question ou le nombre de fois où elles ont été modifiées. Et si nous voulions par exemple loguer également les adresses auxquelles la VM a juste

<sup>4</sup>Elle est matérialisée par un *radix tree* de profondeur 4; un arbre dont chaque branche a trois noeuds et une feuille terminale. Ce sont les feuilles terminales qui contiennent les bits.



accédé, il nous serait impossible de les consigner avec cette *bitmap* dans sa structure actuelle. D'où les limites 3&5 auxquelles cette modification vient remédier.

Nous proposons de modifier l'actuelle *bitmap* de sorte que ses feuilles terminales puissent contenir les informations suivantes :

- Les gPAs loguées dans le *pml\_log* : ainsi, lorsque le *pml\_log* sera plein, chacune des adresses qu'il contient sera reportée dans la structure de données.
- Un compteur pour chaque gPA : qui représentera le nombre de fois où l'adresse a été modifiée ou accédée. Ainsi, à chaque événement *pml buffer full*, le compteur de chaque adresse présente dans le *pml\_log* sera incrémentée. Ceci nous permettra de connaître la chaleur des pages.

Ainsi, la structure de données de consolidation des logs ne sera plus constituée de bits mais de nombres sous forme de *long*<sup>5</sup>, d'où la nouvelle appellation *longmap* comme sur la figure 4.2.

## 3.2 Algorithmique d'estimation

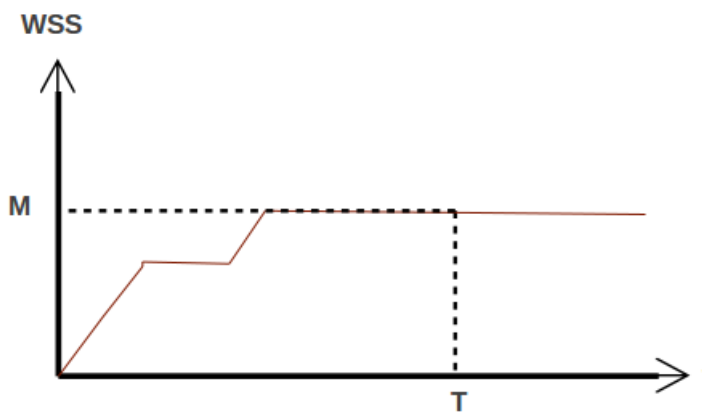


Figure 3.5: Variation de la taille de logs en fonction du temps

Avec le mécanisme de PML, l'algorithme consiste à collecter les logs de la VM, et connaissant les adresses des pages qu'elle utilise, déterminer la quantité de mémoire active

<sup>5</sup>Le type long, comme integer, ou double.

dont elle a effectivement besoin. Le besoin qui se pose, est de savoir le temps pendant lequel il faut collecter. Soit la courbe 3.5 représentant une variation de la taille de logs en fonction du temps. Lorsqu'on commence à observer une constance dans les logs :

- On arrête la collecte et on estime la taille du *working set* en fonction du nombre d'adresses collectées. La taille d'une page étant de  $4Ko$ , si on a collecté  $n$  adresses, alors on estime le WSS à  $M = n * 4Ko$ .
- On applique l'estimation faite à la VM et pour en être sûr on observe son comportement sur une certaine période.
- Si on observe trop de swap ou de défauts de page, alors on effectue de nouveau la collecte et on adapte l'estimation en fonction des observations précédentes.

Seulement la plus part des applications n'ont pas une activité constante.

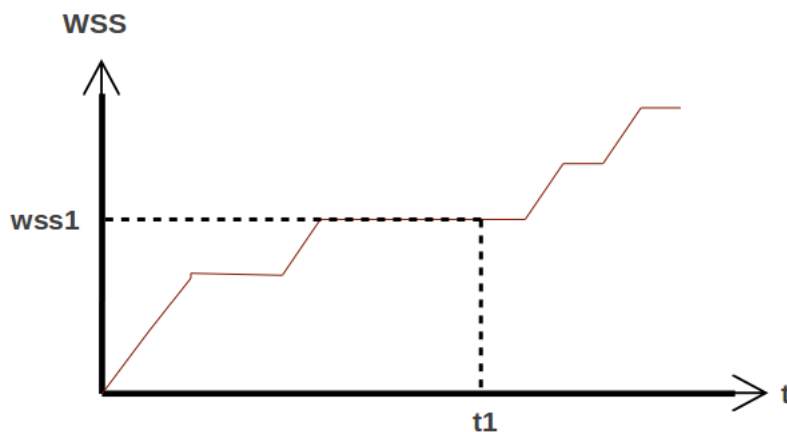


Figure 3.6: Variation de la taille de logs en fonction du temps

Le *working set* de la VM peut être très variable (comme sur la figure 3.6) ou même périodique. Dans la majeure partie des cas, des algorithmes de *machine learning* seront nécessaires pour apprendre le comportement de la VM et savoir quand et comment ajuster sa mémoire.

Sur la figure 3.6, en se basant sur l'algorithme naïf présenté ci-haut, on arrêterait la collecte à  $t1$  et on estimerait le WSS de la VM à  $wss1$ , ce qui est loin d'être sa valeur exacte. Et en réajustant la VM à cette quantité de mémoire, on la mettrait dans un état



de souffrance (la VM sera compressée ce qui conduira à trop de swaps et de défauts de pages).

### 3.3 Avantages de la solution proposée

Nous avons vu dans le chapitre précédent que toute solution d'estimation du WSS devait répondre à deux questions :

- **(Q1) : comment observer la VM et collecter les informations sur son activité?**

La description du mécanisme du PML répond à cette question. Ici, c'est le matériel lui même qui se charge de collecter pour nous les informations sur l'activité mémoire de la VM, en traquant les pages utilisées par la VM et en consignnant les adresses de celles-ci dans une structure de consolidation de logs. Ce qui rend donc notre solution totalement **transparente** du point de vue de la VM, i.e. **non intrusive**.

Actuellement, le coût facturé à la VM concerne l'ensemble des *VMExits* qui lui sont imposés lorsque le *pml\_log* est plein. La solution architecturale que nous proposons vient pallier cette limite en redirigeant ces interruptions vers le dom0, donc **aucune surcharge n'est imposée ni à la VM ni à l'hyperviseur**.

- **(Q2) : comment estimer le *working set* de la VM à partir des données collectées?**

Une fois que l'on connaît toutes les pages utilisées par la VM, il nous suffit de les compter et connaissant la taille <sup>6</sup> d'une page, il n'y a plus qu'à effectuer le calcul présenté à la section 3.2 précédente.

Etant donné que c'est le matériel lui-même qui se charge de récupérer les adresses des pages utilisées par la VM, on peut être sûr qu'aucune adresse ne sera manquée, et donc de la précision de l'estimation. Pour l'instant, cette précision de calcul ne

---


<sup>6</sup>Elle peut dépendre de l'architecture de la machine hôte.

---

peut être vérifiée car il faudrait d'abord que la nouvelle architecture soit mise sur pied.

Après avoir expliquer le mécanisme de fonctionnement actuel du PML, nous avons ressorti dans ce chapitre les cinq principales limites qu'il présente pour l'estimation du WSS. Après quoi, nous avons présenté les améliorations que nous proposons pour concevoir un design architectural du PML qui serait plus adapté pour une technique d'estimation du WSS.

## IMPLÉMENTATION ET ÉVALUATIONS DE LA TECHNIQUE ÉLABORÉE

e chapitre présente quelques détails d'implémentation du prototype réalisé et les évaluations que nous avons effectuées pour valider l'intérêt de notre solution.



## 4.1 Implémentation

### 4.1.1 Environnement d'implémentation

#### 4.1.1.1 Hyperviseur

Plusieurs systèmes de virtualisation existent sur le marché. Nous nous sommes donc appuyés sur un certain nombre de critères pour choisir celui que nous allons utiliser dans notre travail. Il s'agit :

- ☞ Du **support para-virtualisation**. La solution que nous proposons concorde avec les principes de la para-virtualisation car elle exige des modifications dans les systèmes d'exploitation invités.
- ☞ Du **droit de modification des sources**. L'hyperviseur choisi doit être *open source* afin de pouvoir en obtenir le code source et de pouvoir le modifier sans contrainte.
- ☞ De la **communauté informatique travaillant sur le système**. À cause de la documentation disponible et du support à la résolution des bugs.

Sur la base de ces critères, nous avons opté pour l'utilisation de l'hyperviseur *XEN* [9] car c'est celui qui correspondait le mieux à nos besoins.

#### 4.1.1.2 Système d'exploitation

Les critères de choix du système d'exploitation sont les mêmes que ceux utilisés pour le choix de l'hyperviseur, avec en plus la contrainte de compatibilité entre les deux. Le choix n'a donc pas été difficile, nous avons opté pour le système d'exploitation *Linux* car:

- ☞ Linux est le seul système d'exploitation qui supporte la para-virtualisation avec l'hyperviseur Xen;
- ☞ Linux est *open source* : son code source est disponible et sa modification n'est pas interdite;



- ☞ Il y a une grande communauté qui travaille sur ce système. En cas de problème, il y a assez de documentations et de forums qui pourront permettre de trouver des solutions aux problèmes rencontrés.

#### 4.1.1.3 Langage de programmation

Les mécanismes que nous voulons implémenter nécessitent d'interagir avec le matériel. Il est donc important d'utiliser un langage de bas niveau. De plus, il s'agit essentiellement de modifier les codes sources de l'hyperviseur et du système d'exploitation, qui sont tous les deux écrits en langage C. C'est la raison pour laquelle nous avons utilisé le C comme langage de programmation.

#### 4.1.1.4 Outils de développement

##### Le compilateur

Le noyau Linux et l'hyperviseur XEN sont écrits dans le langage de programmation C, avec certaines parties du code en langage assembleur. Pour les compiler, il faut utiliser la suite de compilateurs *GCC* (GNU Compiler Collection). *GCC* est un ensemble de compilateurs créés par le projet GNU [28]. *GCC* est un logiciel libre capable de compiler divers langages de programmation, dont C, C++, Objective-C, Java, Ada et Fortran.

On a besoin en outre de l'outil *make* qui est un programme permettant de parcourir le code source et de déterminer quels sont les fichiers qui ont besoin d'être compilés, afin d'appeler le compilateur *GCC* pour faire le reste du travail.

##### L'éditeur

Lorsqu'on développe sur des projets contenant des millions de lignes de code comme ceux de Linux (environ 16 millions de lignes de code réparties dans 45 mille fichiers) [18] et XEN (environ 600 mille lignes de code réparties dans 7 mille fichiers), il est nécessaire d'utiliser un bon éditeur pour faciliter la navigation entre les fichiers. Nous avons opté pour les éditeurs *VS Code*, *Visual Studio Code* et *Sublime Text*.



*VS Code* est un éditeur multi-plateforme, *open source* et gratuit supportant une dizaine de langages. C'est un éditeur de code extensible développé par Microsoft pour Windows, Linux et macOS.

VS Code combine la simplicité d'un éditeur de code avec ce dont les développeurs ont besoin pour leur cycle de base édition-compilation-débugage. Le code fournit une prise en charge complète de l'édition et du débogage, un modèle d'extensibilité et une intégration légère aux outils existants. Il permet aussi, entre autres fonctionnalités, de retrouver la définition d'une fonction à partir de l'endroit où elle est appelée et de rechercher et remplacer des chaînes de caractères sur la base d'expressions régulières.

VS Code est mis à jour tous les mois avec de nouvelles fonctionnalités et corrections de buggs.

*Sublime Text* est un éditeur de texte générique codé en C++ et Python, disponible sur Windows, Mac et Linux. Le logiciel a été conçu tout d'abord comme une extension pour Vim, riche en fonctionnalités. Cet éditeur prend en charge plus de 44 langages de programmation majeurs, et des plugins sont souvent disponibles pour les langages plus rares. Sur son interface graphique, il intègre la plupart des fonctionnalités de base d'un éditeur de texte, dont la coloration syntaxique personnalisable et l'auto complétion. Tout comme VS Code, il permet également de retrouver la définition d'une fonction à partir de l'endroit où elle est appelée et de rechercher et remplacer des chaînes de caractères sur la base d'expressions régulières.

## 4.1.2 Détails d'implémentation

### 4.1.2.1 Définition des hypercalls pour activer et désactiver le PML pour une VM donnée

Le mécanisme du PML n'est pas activé par défaut, même s'il est supporté par le processeur. Certains hyperviseurs tels que VMWare et XEN ont déjà modifié leur code source pour y intégrer ce mécanisme, notamment son activation et son utilisation.

XEN dans son code source a intégré le PML, mais uniquement dans le cadre de la migra-



tion des machines virtuelles [32]. En effet, pendant la migration d'une VM, XEN utilise le mécanisme du PML pour traquer les pages chaudes qui doivent être déplacées.

Or dans le cadre de notre travail, nous devons être capables d'activer le PML hors de ce contexte, i.e. chaque fois que nous avons besoin d'estimer le WSS d'une VM. Dans XEN, la figure 4.1 présente la file d'appels des fonctions liées à l'activation du PML pour une VM :

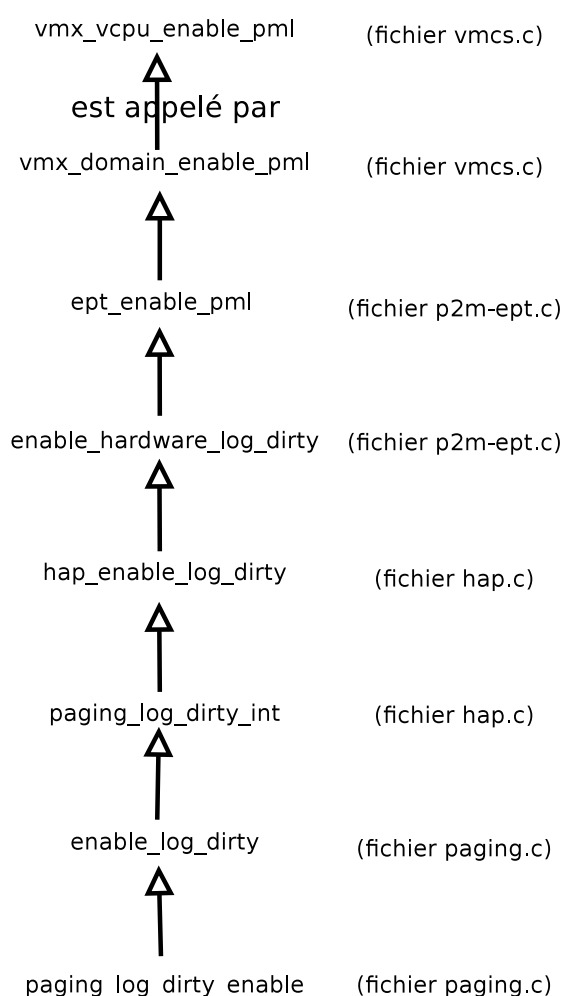


Figure 4.1: File d'appels des fonctions liées à l'activation du PML

Comme nous le montre cette file d'appels, le but est d'activer le PML pour les processeurs sur lesquels s'exécutent la VM. Mais tout part de l'activation du mode *log dirty* pour les pages de la VM. C'est ce mode qui va permettre de traquer les pages mémoire utilisées par la machine virtuelle.



L'activation du mode *log dirty* dans XEN se fait depuis les tools. Nous avons donc défini des hypercalls à travers les commandes suivantes : «*xl enable-log-dirty*» & «*xl disable-log-dirty*».

Un aperçu des lignes de code liées à la définition de ces hypercalls se trouve en annexes 4.2.3, codes 4.1 et 4.2.

#### 4.1.2.2 Modification de la structure de données bitmap

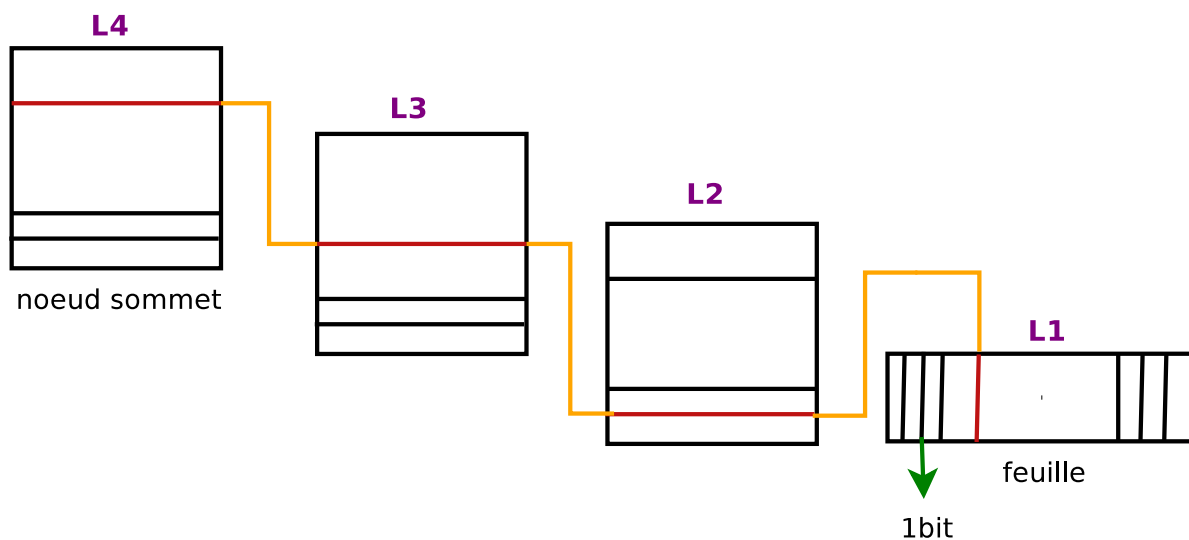


Figure 4.2: Radix tree représentant la bitmap

Actuellement la structure de données qui représente la bitmap est un *radix tree* dont chaque branche est constituée de 4 niveaux tels que matérialisés par la figure 4.2:

- 3 noeuds intermédiaires : dans chaque noeud, une entrée est un pointeur vers une entrée dans le noeud suivant.
- 1 feuille terminale : ses entrées quant à elle sont des bits dont chacun représente une adresse du log.

Chaque noeud de l'arbre est une page de 4Ko, y compris les feuilles terminales. Étant donné que les entrées d'une feuille sont des bits, on peut déterminer le nombre d'entrées dans une feuille. En effet :

$$1\text{octet} = 2^3\text{bits}$$



$$\Rightarrow 1Ko = 2^{10} * 2^3 bits$$

$$\Rightarrow 4Ko = 2^2 * 2^{10} * 2^3 bits = 2^{15} bits$$

Donc on a  $2^{15}$  entrées dans une feuille. Maintenant nous devons garder pour une feuille le même nombre d'entrées mais de type *long*<sup>1</sup> et non plus *bit*. Pour cela nous devons déterminer le nombre  $X$  de pages de 4Ko dont a besoin pour contenir  $2^{15}$  entrées de type *long* :

$$\begin{aligned} X &= \frac{taille\_des\_entrees}{taille\_page} \\ &= \frac{2^{15} * sizeof(UL)}{4Ko} \\ &= \frac{2^{15} * sizeof(UL)^2}{2^2 * 2^{10}} \\ &= 2^3 * sizeof(UL) \\ &= 8sizeof(UL) \end{aligned}$$

Sur la base de ces calculs, nous avons rajouté un niveau  $L0$ . Le *radix tree* est donc modifié de sorte que les entrées du niveau  $L1$  actuel soient non plus des bits mais des adresses qui pointent chacune vers une page  $L0$ , qui elle contiendra les informations loguées i.e. adresses et occurrences.

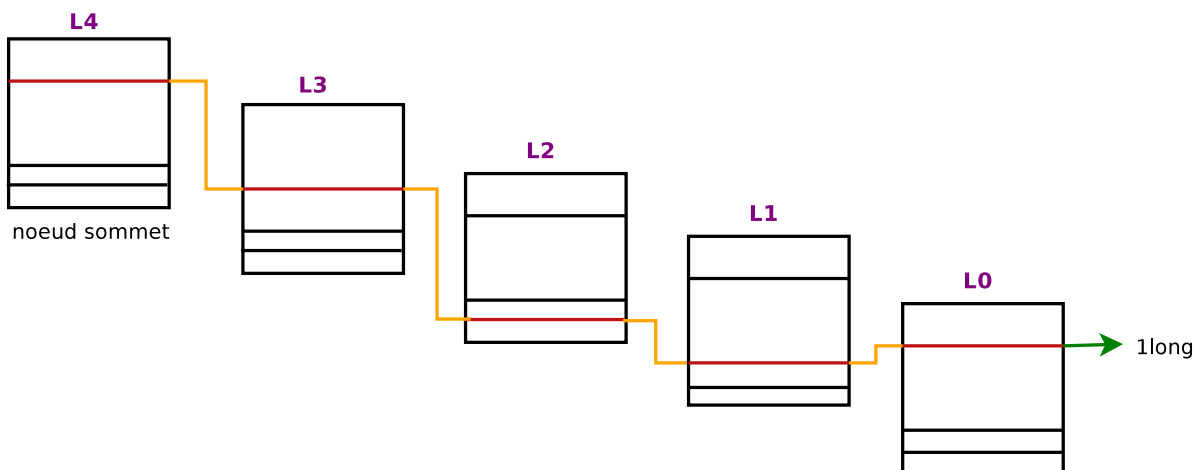


Figure 4.3: Radix tree représentant la longmap

<sup>1</sup>UL : unsigned long



La méthode qui permettait de générer une feuille *L1* a été modifiée telle que présentée dans les codes 4.3 et 4.4 en annexe 4.2.3. Et la nouvelle structure est matérialisée par la figure 4.3.

#### 4.1.2.3 Modification du traitant *pml buffer full*

Nous avons modifié le comportement du mécanisme lorsque le *pml\_log* est plein en accord avec la modification de la structure de consolidation de logs telle que présentée au paragraphe précédent. Ainsi, au moment de vider le buffer, l'algorithme parcourt la nouvelle structure de données, et s'il existe déjà une entrée pour l'adresse à enregistrer, son compteur est incrémenté. Sinon, une nouvelle entrée est créée et le compteur est initialisé à 1 pour cette adresse.

Les modifications effectuées sont dans les codes 4.5 et 4.6 en annexe 4.2.3.

#### 4.1.2.4 Mise en place du mécanisme (hypercall) de copie des logs consolidés de l'hyperviseur vers le dom0

Pour éviter d'imposer des coûts de surcharge à l'hyperviseur, nous implémentons les algos de calcul dans le dom0. Pour cela, il faut copier les logs depuis la longmap dans l'hyperviseur, vers le dom0 (figure 4.4). Ceci ne peut se faire qu'à l'aide d'un hypercall que nous avons donc défini : «*xl collect - dirty - logs*».

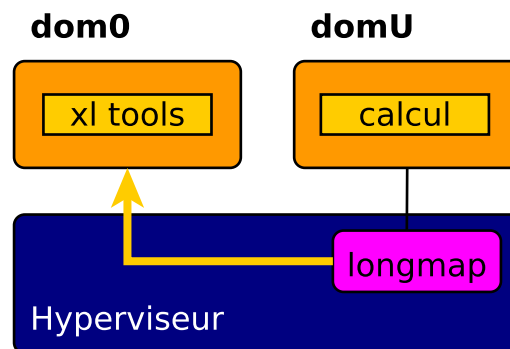


Figure 4.4: Copie des logs de l'hyperviseur vers le dom0



Cet hypercall est appelé avec l'*id* de la VM en paramètre. Ainsi, dès qu'on fait un «*xl collect-dirty-logs id\_vm*», la chaîne d'appels de fonctions de l'hypercall est la suivante :

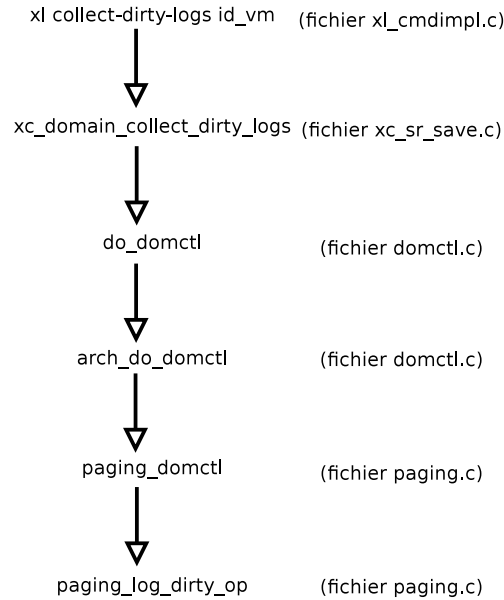


Figure 4.5: Copie des logs de l'hyperviseur vers le dom0 : chaîne d'appel de fonctions

Cette dernière méthode *paging\_log\_dirty\_op*, nous l'avons modifiée pour remonter les logs vers le dom0. L'algorithme implémenté dans cette méthode consiste à copier dans les logs de XEN, le contenu de la *longmap* et compter le nombre total de pages différentes.

La définition de la commande *xl* pour la collecte est en annexe 4.2.3, code 4.9, et celle de la fonction *xc\_domain\_collect\_dirty\_logs* code 4.8.

#### 4.1.2.5 Implémentation de la version 0 de l'algorithme d'estimation du WSS

Enfin, nous avons implémenter une première version de l'algorithme que nous présentons au chapitre précédent. Cette version ne prend en compte que les parties de l'architecture proposée que nous avons pu implémenter, les autres nécessitant une modification du matériel (modification du processeur).

Nous avons écrit un script (voir code 4.10) qui permet de collecter les logs d'une VM sur une période  $T$ , ceci toutes  $x$  les secondes ( $T$  et  $x$  donnés en paramètre du script). Le

script trace au fur et à mesure de la collecte, une courbe représentant la variation du nombre total d'adresses enregistrées en fonction du temps. Lorsque cette variation se stabilise, i.e. lorsque la courbe devient constante, l'algorithme s'arrête et l'estimation est faite.

Des exemples d'application de cet algorithme et des graphes générés sont présentés ci-après dans la section *Évaluations*.

## 4.2 Évaluations

### 4.2.1 Environnement d'évaluation

Les expérimentations que nous avons menées ont nécessité l'utilisation d'un seul serveur physique. Les caractéristiques de la machine hôte sont regroupées dans le tableau 4.1 suivant.

Tableau 4.1: Caractéristiques de la machine physique dans les expérimentations

Caractéristique	Valeur
<b>Fabricant</b>	Dell Inc
<b>Famille de processeur</b>	Intel(R) Core(TM) i7-7600U
<b>Architecture du processeur</b>	x86_64
<b>Nombre de CPUs</b>	4
<b>Nombre de cœurs par CPU</b>	2
<b>Fréquence des cœurs</b>	2,80GHz
<b>Taille de la RAM</b>	16Go

Nous avons effectué les expérimentations avec des charges de travail synthétiques.

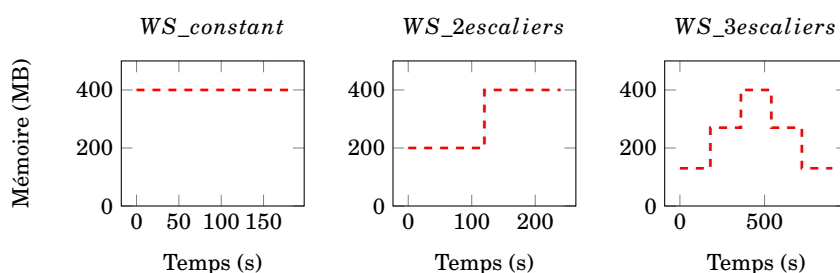


Figure 4.6: Ensemble des charges de travail synthétiques



La figure 4.6 présente pour chacune des charges, la variation attendue du WSS en fonction du temps. Nous avons exécuté dans la VM des applications qui manipulent toutes un *working set* de  $400MB$  (soit  $400 * 1024/4 = 102400$  pages mémoire), mais de façons différentes : utilisation constante et variation en escaliers.

Pour chacune des expérimentations ci-après, nous allons présenter les résultats obtenus avec les solutions existantes et ensuite avec la solution que nous avons implémentée.

**Sur les graphes ci-après présentant les résultats d'expérimentation,  $W^o$  représente le *working set attendu* (courbe en rouge) et  $W^e$  le *working set estimé* (courbe en bleu).**

#### 4.2.2 Expérimentation 1 : charge synthétique manipulant un *working set* constant

Comme nous pouvons le voir sur la figure 4.8, la technique que nous avons mise sur pied détecte 105000 pages, soit  $410MB$  comme valeur estimée du *working set*, pour un programme manipulant 102400 pages.

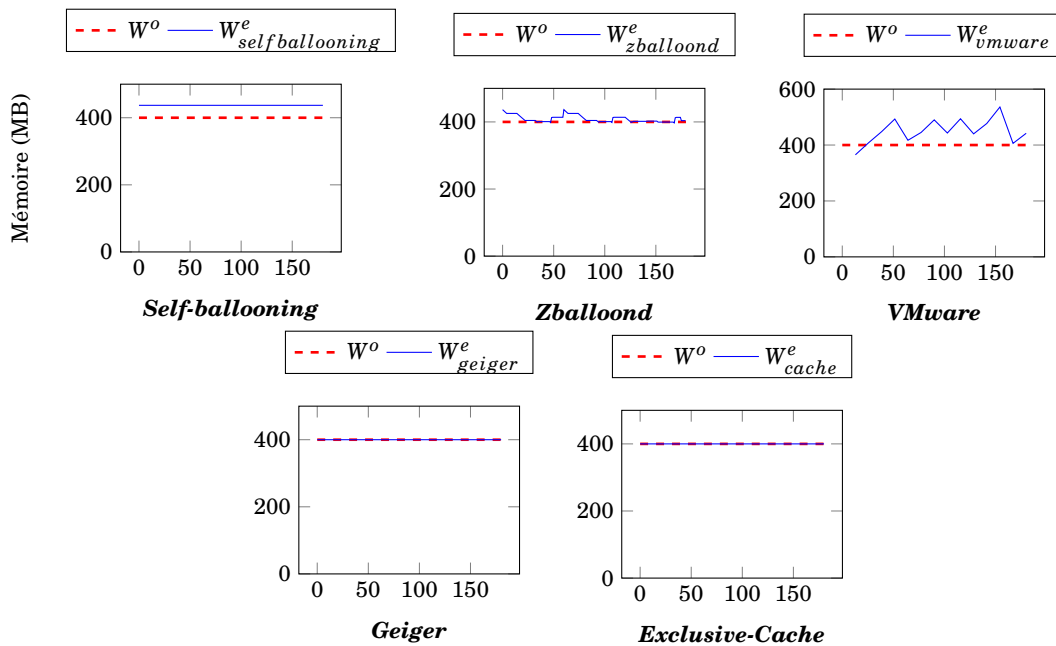


Figure 4.7: Résultats d'évaluations de la première charge synthétique avec les techniques existantes



L'application qui tourne dans la VM utilise activement les 102400 pages qui constituent son *working set*. Le surplus observé trouve son explication au paragraphe 3.1.1.2 - *limite* 4. En effet, comme nous l'avons expliqué les pages de la table de pages sont également loguées pendant le mécanisme, et celles-ci ne sont pas prises en compte dans la valeur expérimentale de la mémoire utilisée par l'application.

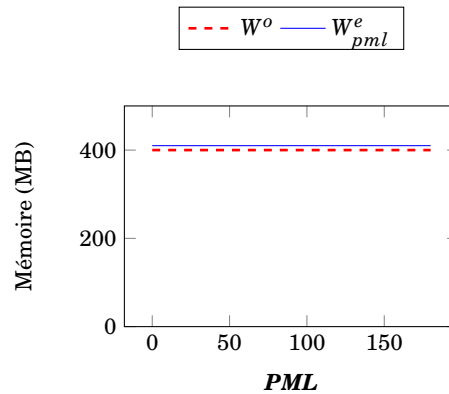


Figure 4.8: Résultats d'évaluations de la première charge synthétique avec la technique implémentée

Cette première expérimentation montre que notre solution estime correctement le WSS avec des charges constantes. Par contre, la plupart des techniques existantes passent à côté de l'estimation. Seules deux techniques, Geiger et Exclusive-cache parviennent à donner une estimation précise, mais avec les limites mentionnées dans le chapitre 2.

### 4.2.3 Expérimentation 2 : charges synthétiques manipulant un *working set variable* (en escaliers)

En observant les résultats présentés à la figure 4.9, on constate que pour la plupart des techniques existantes il est difficile de détecter les variations de mémoire, à l'opposé de la technique que nous avons implémentée dont les résultats sont présentés à la figure 4.10.



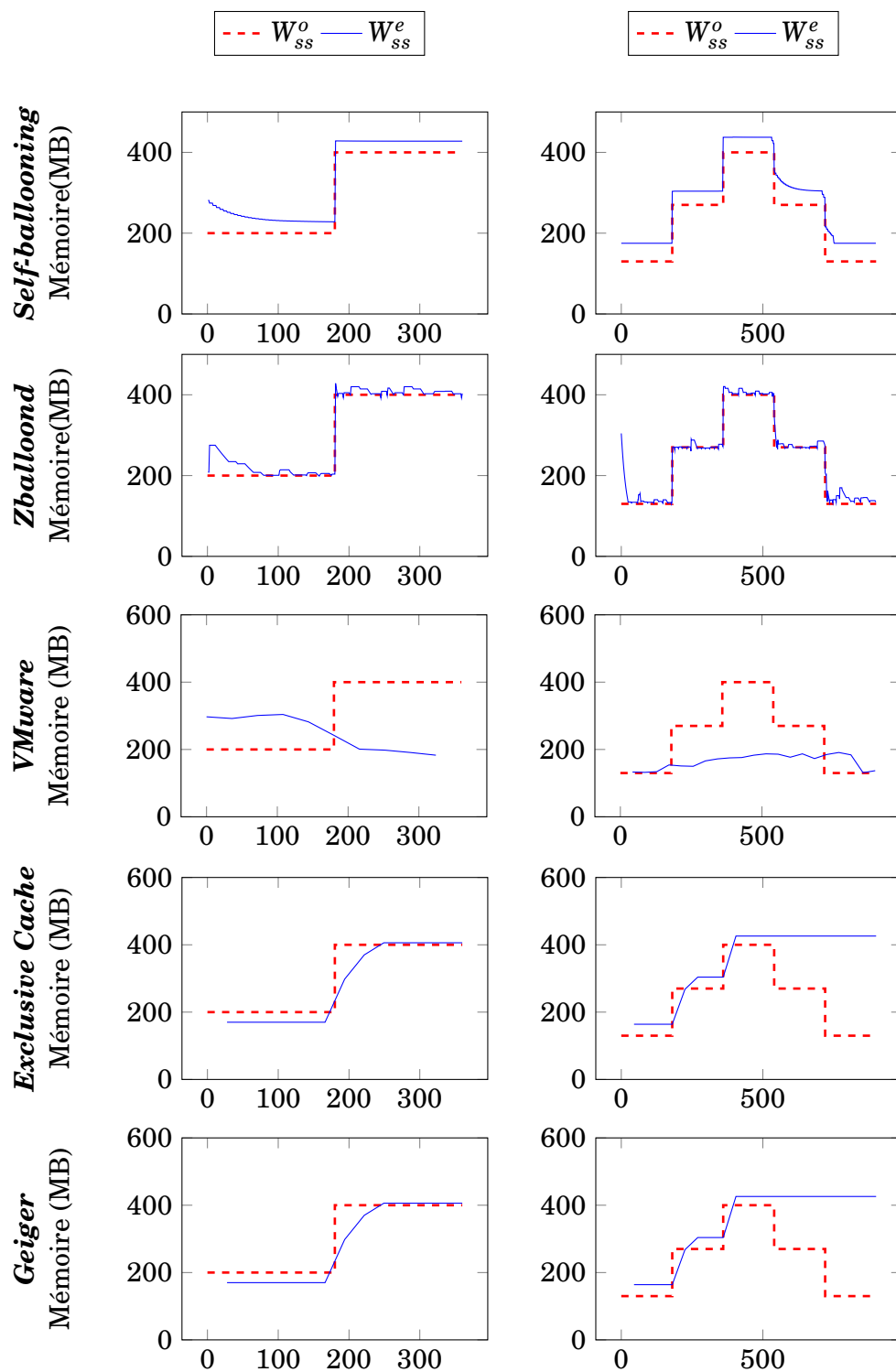


Figure 4.9: Résultats d'évaluations des charges variables avec les techniques existantes

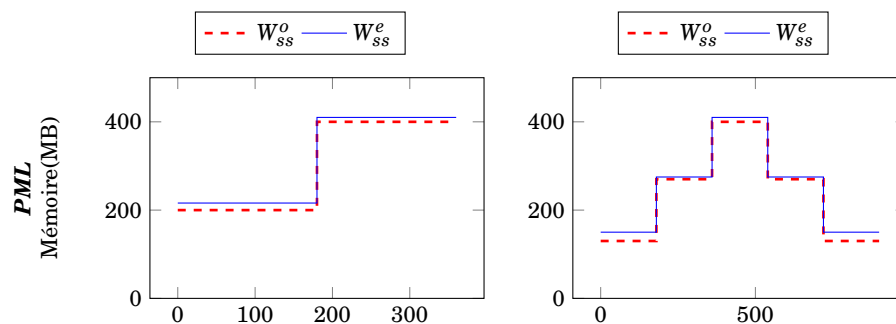


Figure 4.10: Résultats d'évaluations des charges variables avec le PML

Dans ce chapitre, nous présentons le prototype en environnement réel que nous avons implémenté, et qui permettra d'exploiter le nouveau design que nous avons proposé en contributions. Ce prototype, nous l'avons testé avec le mécanisme du PML tel que fonctionnant actuellement, et les résultats sont présentés ici en comparaison avec les solutions existantes.

# CONCLUSION

## Bilan

Dans ce mémoire nous avons présenté une solution au problème d'estimation du *working set*, solution qui se base sur les améliorations des processeurs à ce sujet.

Pour cela, nous avons pris appui sur une amélioration matérielle des processeurs Intel : le PML (Page Modification Logging). Après avoir fait un rappel des concepts nécessaires à la compréhension du sujet abordé, nous avons fait le tour des solutions déjà établies pour répondre au même problème. Par la suite, nous avons exposé les limites de cette fonctionnalité matérielle ainsi que les améliorations architecturales que nous proposons suite à ces limites. Et pour finir, nous avons fait part de notre solution à travers les détails d'implémentation et les résultats des tests effectués.

Les résultats des expérimentations montrent que notre solution est un bon estimateur comparé à la plus part des autres solutions existantes. En outre, la technique implémentée permet de détecter les changements dans l'utilisation de mémoire par la machine.

Le tableau 4.2 récapitule les critères d'estimation de notre solution en comparaison avec ceux des techniques existantes.

Tableau 4.2: Tableau comparatif des techniques d'estimation du WSS existantes

Technique	Intrusive	Active	Précise	Surcharge la VM	Surcharge l'hyperviseur
Self-ballooning	Oui	Non	Non	Non	Non
ZBalloond	Oui	Oui	Non	Oui	Non
VMWare	Non	Oui	Non	Non	Oui
Geiger	Non	Non	Non si $WSS < \text{mémoire allouée}$	Non	Non
Exclusive-cache	Non	Oui	Non si le cache est nul	Non	Non
PML	Non	Non	Oui (avec le nouveau design)	Sous réserve d'un simulateur	

## Conclusion



Toutefois, cette technique est loin d'être à point étant donné les modifications matérielles qu'elle nécessite.


## Perspectives

Nous n'avons pu implémenter que partiellement la solution que nous proposons. Nous envisageons donc pour de futures échéances :

- ☛ L'optimisation de la technique d'estimation.
- ☛ Le développement d'un simulateur pour tester les performances de la nouvelle architecture que nous proposons (redirection des interruptions vers le dom0, élimination des adresses des pages de la table de page, utilisation de deux buffer pour la collecte des logs).

## RÉFÉRENCES

- [1] [Consulté en mai 2018]. URL: <http://www.intel.com/design/literature.https://blog.xenproject.org/2008/08/27/xen-33-feature-memoryovercommit/>.
- [2] [Consulté en mai 2018]. URL: [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/6/html/deployment\\_guide/s2-proc-meminfo..](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-meminfo..)
- [3] [Consulté en mai 2018]. URL: [https://www.linuxquestions.org/questions/linux-server-73/committed\\_as-in-proc-meminfo-4175595916/](https://www.linuxquestions.org/questions/linux-server-73/committed_as-in-proc-meminfo-4175595916/).
- [4] F. Aderholdt et al. “Efficient Checkpointing of Virtual Machines Using Virtual Machine Introspection”. In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2014, pp. 414–423. DOI: 10.1109/CCGrid.2014.72.
- [5] Sherif Akoush et al. “Predicting the Performance of Virtual Machine Migration”. In: *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. MASCOTS ’10 (2010), pp. 37–46. DOI: 10.1109/MASCOTS.2010.13. URL: <https://doi.org/10.1109/MASCOTS.2010.13>.
- [6] Hanna Alam et al. “Do-It-Yourself Virtual Memory Translation”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 457–468. ISSN: 0163-5964. DOI: 10.1145/3140659.3080209. URL: <http://doi.acm.org/10.1145/3140659.3080209>.

- 
- 
- 
- [7] Amazon Web Services Inc. [Consulté le 12 Juillet 2018]. URL: <https://aws.amazon.com/ec2/>.
  - [8] Arpaci-Dusseau, Remzi H., Arpaci-Dusseau, Andrea C. *Operating Systems: Three Easy Pieces, [Chapter: Faster Translations (TLBs)]*. Arpaci-Dusseau Books, 2014.
  - [9] Paul Barham et al. “Xen and the Art of Virtualization”. In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: <http://doi.acm.org/10.1145/1165389.945462>.
  - [10] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan and Claypool Publishers, 2009. ISBN: 159829556X, 9781598295566.
  - [11] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. “Working Set-based Physical Memory Ballooning”. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 95–99. ISBN: 978-1-931971-02-7. URL: <https://www.usenix.org/conference/icac13/technical-sessions/presentation/chiang>.
  - [12] Christopher Clark et al. “Live Migration of Virtual Machines”. In: *2Nd Conference on Symposium on Networked Systems Design & Implementation. NSDI’05 Volume 2* (2005), pp. 273–286. URL: <http://dl.acm.org/citation.cfm?id=1251203.1251223>.
  - [13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Vol. 3. Mars 2018. ISBN: 325462-066US.
  - [14] *e-Energy ’11: Proceedings of the 2Nd International Conference on Energy-Efficient Computing and Networking*. New York, New York, USA: ACM, 2011. ISBN: 978-1-4503-1313-1.
  - [15] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. “Agile Paging: Exceeding the Best of Nested and Shadow Paging”. In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 707–718. ISSN: 0163-5964. DOI: 10.1145/3007787.3001212. URL: <http://doi.acm.org/10.1145/3007787.3001212>.




- [16] Gartner Inc. *Technologie de virtualisation et logiciels de VM*. [Consulté le 18 Juillet 2018]. 2016. URL: <https://www.vmware.com/fr/solutions/virtualization.html>.
- [17] Intel Corporation. *Page Modification Logging for Virtual Machine Monitor White Paper*. [Consulté le 10 mars 2018]. Janvier 2015. URL: <http://www.intel.com/design/literature.htm>.
- [18] Paul Jones. *Linux: The Fundamentals Of The Linux Operating System A Complete Beginners Guide To Linux Mastery*. USA: CreateSpace Independent Publishing Platform, 2017. ISBN: 1544653085, 9781544653082.
- [19] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment”. In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006), pp. 14–24. ISSN: 0163-5964. DOI: 10.1145/1168919.1168861. URL: <http://doi.acm.org/10.1145/1168919.1168861>.
- [20] Jinchun Kim et al. “Dynamic Memory Pressure Aware Ballooning”. In: *Proceedings of the 2015 International Symposium on Memory Systems*. MEMSYS '15. Washington DC, DC, USA: ACM, 2015, pp. 103–112. ISBN: 978-1-4503-3604-8. DOI: 10.1145/2818950.2818967. URL: <http://doi.acm.org/10.1145/2818950.2818967>.
- [21] Pin Lu and Kai Shen. “Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache”. In: *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. ATC'07. Santa Clara, CA: USENIX Association, 2007, 3:1–3:15. ISBN: 999-8888-77-6. URL: <http://dl.acm.org/citation.cfm?id=1364385.1364388>.
- [22] Vikas Malik and C. R. Barde. “Article: Live Migration of Virtual Machines in Cloud Environment using Prediction of CPU Usage”. In: *International Journal of Computer Applications* 117.23 (May 2015). Full text available, pp. 1–5.



- [23] Sparsh Mittal. “A Survey of Techniques for Architecting and Managing GPU Register File”. In: *IEEE Trans. Parallel Distrib. Syst.* 28.1 (Jan. 2017), pp. 16–28. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2546249. URL: <https://doi.org/10.1109/TPDS.2016.2546249>.
- [24] Vlad Nitu et al. “Working Set Size Estimation Techniques in Virtualized Environments: One Size Does Not Fit All”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 2.1 (Apr. 2018), 19:1–19:22. ISSN: 2476-1249. DOI: 10.1145/3179422. URL: <http://doi.acm.org/10.1145/3179422>.
- [25] R. H. Patterson et al. “Informed Prefetching and Caching”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP ’95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 79–95. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224064. URL: <http://doi.acm.org/10.1145/224056.224064>.
- [26] Wing-Chi Poon and Aloysius K. Mok. “Improving the Latency of VMExit Forwarding in Recursive Virtualization for the x86 Architecture”. In: *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*. HICSS ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 5604–5612. ISBN: 978-0-7695-4525-7. DOI: 10.1109/HICSS.2012.320. URL: <https://doi.org/10.1109/HICSS.2012.320>.
- [27] Primalmotion. *User contributions for Primalmotion*. Aug. 2008. URL: <https://commons.wikimedia.org/wiki/Special:Contributions/Primalmotion>.
- [28] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. ISBN: 144141276X, 9781441412768.
- [29] *Translation lookaside buffer*. [Consulté le 13 avril 2018]. 25 octobre 2017. URL: [https://fr.wikipedia.org/wiki/Translation\\_lookaside\\_buffer](https://fr.wikipedia.org/wiki/Translation_lookaside_buffer).
- [30] VMware Inc. *Gartner Says Worldwide Server Virtualization Market Is Reaching Its Peak*. [Consulté le 18 Juillet 2018]. 2017. URL: <http://www.gartner.com/newsroom/id/3315817>.



- 
- 
- 
- [31] Carl A. Waldspurger. “Memory Resource Management in VMware ESX Server”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146. URL: <http://doi.acm.org/10.1145/844128.844146>.
- [32] Kai Huang in Xen-devel mailing list. *PML (Page Modification Logging) design for Xen*. [Consulté le 11 avril 2018]. Février 2015. URL: <https://lists.xenproject.org/archives/html/xen-devel/2015-02/msg01305.html>.

## ANNEXES

### Annexe 1 : Description des structures d'accueil IRIT et LaBRI

**L**e travail présenté dans ce document a été réalisé en collaboration avec les laboratoires IRIT (Institut de Recherche en Informatique de Toulouse) et LaBRI (Laboratoire Bordelais de Recherche en Informatique).

L'IRIT est l'une des plus imposantes Unités Mixtes de Recherche (UMR) au niveau national (français), et l'un des piliers de la recherche en Midi-Pyrénées avec ses 700 membres permanents et non-permanents. De par son caractère multi-tutelle (CNRS, INPT, Universités toulousaines), son impact scientifique et ses interactions avec les autres domaines, l'IRIT constitue une des forces structurantes du paysage de l'informatique et de ses applications dans le monde du numérique, tant au niveau régional que national. Cette unité est, depuis 2003, structurée en 7 thèmes de recherche qui regroupent les 21 équipes du laboratoire :

- ☞ Analyse et synthèse de l'information (4 équipes);
- ☞ Indexation et recherche d'informations (3 équipes);
- ☞ Interaction, Coopération, Adaptation par l'Expérimentation (2 équipes);
- ☞ Raisonnement et décision (3 équipes);
- ☞ Modélisation, algorithmes et calcul haute performance (1 équipe);
- ☞ Architecture, systèmes et réseaux (5 équipes);



☞ Sûreté de développement du logiciel (3 équipes).

Le laboratoire travaille à instaurer un continuum allant de la recherche à la valorisation, en imaginant et développant des formes de collaboration innovantes avec ses partenaires. La fertilisation du tissu économique local et national via les grands groupes mais aussi en resserrant ses relations avec les PME et PMI, et en s'impliquant dans diverses structures d'animation, sont devenus pour l'IRIT un devoir.

Le LaBRI est une unité de recherche associée au CNRS (UMR 5800), à l'Université de Bordeaux et à Bordeaux INP. Depuis 2002, il est partenaire de l'Inria. Ses effectifs se sont accrus de façon importante ces dernières années. En novembre 2017, il réunit près de 280 personnes, dont 110 enseignants chercheurs (Université de Bordeaux, Bordeaux INP), 41 chercheurs (CNRS, Inria), 20 personnels administratifs et techniques (Université de Bordeaux, Bordeaux INP, CNRS, Inria) et plus de 100 doctorants, post-doctorants et ingénieurs contractuels. Les missions du LaBRI s'articulent autour de trois axes principaux : recherche (théorique, appliquée), valorisation - transfert de technologie et formation.

Le soutien du Conseil Régional d'Aquitaine à travers l'extension du bâtiment, des équipements et des bourses de thèse et post-doctorants, a été une des rubriques essentielles du développement du LaBRI.

Le laboratoire s'articule autour de six équipes thématiques alliant recherche fondamentale, recherche appliquée et transfert technologique :

- ☞ Combinatoire et Algorithmique;
- ☞ Image et Son;
- ☞ Méthodes Formelles;
- ☞ Modèles et Algorithmes pour la Bioinformatique et la Visualisation d'informations;
- ☞ Programmation, Réseaux et Systèmes;
- ☞ Supports et Algorithmes pour les Applications Numériques hautes performances.





Les chercheurs et enseignants-chercheurs du LaBRI participent à la formation initiale et continue dans différents établissements de plus de 1300 étudiants inscrits dans les spécialités informatiques du L (Licence), M (Master) et D (Doctorat) mis en place à la rentrée 2003. A travers des réseaux et structures variés, le LaBRI collabore activement, sur les plans internationaux, européens et français, avec de nombreux laboratoires et entreprises.

## Annexe 2 : Hypercalls d'activation et de désactivation du PML

```
1 int main_enable_log_dirty(int argc, char **argv)
2 {
3     int ret;
4     char path[]="/usr/local/lib/xen/bin/libxl-save-helper";
5     ret=execl(path,path,"--enable-log-dirty",argv[1],NULL);
6     return ret;
7 }
```

Code 4.1: xl enable-log-dirty

```
1 int main_disable_log_dirty(int argc, char **argv)
2 {
3     int ret;
4     char path[]="/usr/local/lib/xen/bin/libxl-save-helper";
5     ret=execl(path,path,"--disable-log-dirty",argv[1],NULL);
6     return ret;
7 }
```

Code 4.2: xl disable-log-dirty

## Annexe 3 : Modification de la structure de données bitmap





```

1  /* Alloc and init a new leaf node */
2  static mfn_t paging_new_log_dirty_leaf(struct domain *d)
3  {
4      mfn_t mfn = paging_new_log_dirty_page(d);
5
6      if ( mfn_valid(mfn) )
7          clear_domain_page(mfn);
8
9      return mfn;
10 }

```

Code 4.3: Méthode appelée générer une feuille L1 avant modification de la bitmap

```

1  /* Alloc and init a new leaf with xxentries */
2  static mfn_t paging_new_log_dirty_leaf_long(struct domain *d)
3  {
4      mfn_t mfn = paging_new_log_dirty_page(d); //mfn2,
5      if ( mfn_valid(mfn) )
6      {
7          int i;
8          mfn_t *node = map_domain_page(mfn);
9          for ( i = 0; i < LOGDIRTY_LEAF_LONG_ENTRIES << 1; i++ )
10             {
11                 node[i] = _mfn(INVALID_MFN);
12             }
13             unmap_domain_page(node);
14         }
15         return mfn;
16     }

```

Code 4.4: Méthode appelée pour générer un noeud L1 après modification de la bitmap

## Annexe 4 : Modification du traitant *pml buffer full*

```

1  /* Mark a page as dirty, with taking guest pfn as parameter */
2  void paging_mark_gfn_dirty(struct domain *d, unsigned long pfn)
3  {

```





```

4    int changed;
5    mfn_t mfn, *l4, *l3, *l2;
6    unsigned long *l1;
7    int i1, i2, i3, i4;
8
9    if ( !paging_mode_log_dirty(d) )
10        return;
11
12    /* Shared MFNs should NEVER be marked dirty */
13    BUG_ON(SHARED_M2P(pfn));
14
15    /*
16     * Values with the MSB set denote MFNs that aren't really part of the
17     * domain's pseudo-physical memory map (e.g., the shared info frame).
18     * Nothing to do here...
19     */
20    if ( unlikely(!VALID_M2P(pfn)) )
21        return;
22
23    i1 = L1_LOGDIRTY_IDX(pfn);
24    i2 = L2_LOGDIRTY_IDX(pfn);
25    i3 = L3_LOGDIRTY_IDX(pfn);
26    i4 = L4_LOGDIRTY_IDX(pfn);
27
28    /* Recursive: this is called from inside the shadow code */
29    paging_lock_recursive(d);
30
31    if ( unlikely(!mfn_valid(d->arch.paging.log_dirty.top)) )
32    {
33        d->arch.paging.log_dirty.top = paging_new_log_dirty_node(d);
34        if ( unlikely(!mfn_valid(d->arch.paging.log_dirty.top)) )
35            goto out;
36    }
37
38    l4 = paging_map_log_dirty_bitmap(d);

```





```

39     mfn = l4[i4];
40     if ( !mfn_valid(mfn) )
41         l4[i4] = mfn = paging_new_log_dirty_node(d);
42     unmap_domain_page(l4);
43     if ( !mfn_valid(mfn) )
44         goto out;
45
46     l3 = map_domain_page(mfn);
47     mfn = l3[i3];
48     if ( !mfn_valid(mfn) )
49         l3[i3] = mfn = paging_new_log_dirty_node(d);
50     unmap_domain_page(l3);
51     if ( !mfn_valid(mfn) )
52         goto out;
53
54     l2 = map_domain_page(mfn);
55     mfn = l2[i2];
56     if ( !mfn_valid(mfn) )
57         l2[i2] = mfn = paging_new_log_dirty_leaf(d);
58     unmap_domain_page(l2);
59     if ( !mfn_valid(mfn) )
60         goto out;
61     l1 = map_domain_page(mfn);
62     changed = !__test_and_set_bit(i1, l1);
63     unmap_domain_page(l1);
64     if ( changed )
65     {
66         PAGING_DEBUG(LOGDIRTY,
67             "marked mfn %" PRI_mfn " ( pfn=%lx), dom %d\n",
68             mfn_x(mfn), pfn, d->domain_id);
69         d->arch.paging.log_dirty.dirty_count++;
70     }
71
72 out:
73     /* We've already recorded any failed allocations */

```





```

74     paging_unlock(d);
75     return;
76 }

```

Code 4.5: Méthode appelée lors de l'évènement *pml buffer full* avant modification

```

1
2 /* Mark a page as dirty, with taking guest pfn as parameter */
3 void paging_mark_gfn_dirty(struct domain *d, unsigned long pfn)
4 {
5     ....
6
7     l2 = map_domain_page(mfn);
8     mfn = l2[i2];
9     if ( !mfn_valid(mfn) ){
10         l2[i2] = mfn = paging_new_log_dirty_leaf_long(d);
11     }
12     unmap_domain_page(l2);
13     if ( !mfn_valid(mfn) )
14         goto out;
15
16     l1 = map_domain_page(mfn);
17     /*
18     *On obtient le bloc de l1      partir d'o      incr menter
19     */
20     decalage = (i1 >> 9); //On fait i1:(PAGE_SIZE/sizeof(long)) et PAGE_SIZE(==1
21     << 12)/sizeof(long)(== 1 << 3) = (1 << 9)
22     i1 %= (PAGE_SIZE >> 3); //(PAGE_SIZE/sizeof(long))
23
24     mfn=l1[decalage];
25     if ( !mfn_valid(mfn) ){
26         l1[decalage] = mfn = paging_new_log_dirty_leaf(d);
27     }
28     unmap_domain_page(l1);
29     if ( !mfn_valid(mfn) )
30         goto out;

```







```

30
31     l0=map_domain_page(mfn);
32     if(l0)
33     {
34         l0[i1]++;
35         unmap_domain_page(l0);
36     }
37
38     /*
39     *On obtient le bloc de l1  partir d'o  ins rer le pfn pris en
40     param tres
41     */
42     decalage += LOGDIRTY_LEAF_LONG_ENTRIES;
43     mfn=l1[decalage];
44     if ( !mfn_valid(mfn) ){
45         l1[decalage] = mfn = paging_new_log_dirty_leaf(d);
46     }
47     unmap_domain_page(l1);
48     if ( !mfn_valid(mfn) )
49         goto out;
50
51     l0=map_domain_page(mfn);
52     if(l0)
53     {
54         l0[i1] = pfn;
55         unmap_domain_page(l0);
56     }
57     ....
58 }

```

Code 4.6: Portion de code modifiée dans de la méthode appelée lors de l'évènement *pml buffer full*



## Annexe 5 : Mécanisme (hypercall) de copie des logs consolidés de l'hyperviseur vers le dom0

```

1 int main_collect_dirty_logs(int argc, char **argv)
2 {
3     int ret=0;
4     char path[]="/usr/local/lib/xen/bin/libxl-save-helper";
5     ret=execl(path,path,"--collect_dirty_logs",argv[1],NULL);
6     return ret;
7 }

```

Code 4.7: xl collect-dirty-logs

```

1 int xc_domain_collect_dirty_logs(xc_interface *xch, uint32_t dom,
   xc_hypcall_buffer_t *dirty_bitmap)
2 {
3     int rc;
4     unsigned long p2m_size=1045504;
5     xen_pfn_t nr_pfns;
6     unsigned long *db=malloc(sizeof(unsigned long) * p2m_size);
7
8     DECLARE_DOMCTL;
9     DECLARE_HYPERCALL_BOUNCE(db, p2m_size * sizeof(*db),
   XC_HYPERCALL_BUFFER_BOUNCE_OUT);
10    memset(&domctl, 0, sizeof(domctl));
11
12    if(xc_domain_nr_gpfns(xch, (domid_t)dom, &nr_pfns)<0)
13    {
14        PERROR("Unable to obtain p2m_size");
15    }
16
17    domctl.cmd = XEN_DOMCTL_shadow_op;
18    domctl.domain = (domid_t)dom;
19    domctl.u.shadow_op.op = XEN_DOMCTL_SHADOW_OP_PEEK;
20    domctl.u.shadow_op.pages = p2m_size;
21    set_xen_guest_handle(domctl.u.shadow_op.dirty_bitmap, db);

```





```

22
23     rc = do_domctl(xch, &domctl);
24
25     xc_hypcall_bounce_post(xch, db);
26
27     return rc;
28 }

```

Code 4.8: Fonction xc\_domain\_collect\_dirty\_logs

```

1  /* Read a domain's log-dirty bitmap and stats.  If the operation is a CLEAN,
2   * clear the bitmap and stats as well. */
3  static int paging_log_dirty_op(struct domain *d,
4                                struct xen_domctl_shadow_op *sc,
5                                bool_t resuming)
6  {
7      ....
8
9      if(clean)
10     {
11         l0 = ((l1 && mfn_valid(l1[i1])) ? map_domain_page(l1[i1
12         ]) : NULL);
13
14         if (l0)
15         {
16             clear_page(l0);
17             unmap_domain_page(l0);
18         }
19         decalage = i1 + LOGDIRTY_LEAF_LONG_ENTRIES;
20         l0 = ((l1 && mfn_valid(l1[decalage])) ? map_domain_page(
21         l1[decalage]) : NULL);
22
23         if (l0)
24         {
25             clear_page(l0);
26             unmap_domain_page(l0);
27         }
28     } else

```





```

25         {
26             for( i0 = 0; i0 < LOGDIRTY_NODE_ENTRIES; i0++)
27             {
28                 l0 = ((l1 && mfn_valid(l1[i1])) ?
29                     map_domain_page(l1[i1]) : NULL);
30                 if(l0)
31                 {
32                     if(l0[i0]!=0)
33                     {
34                         printk( "(WSS) %lu : ", l0[i0]);
35                     }
36
37                     unmap_domain_page(l0);
38                 }
39
40                 /**
41                  * Ensuite on fait un dcalage pour se placer au
42                  * liste des adresses
43                  */
44                 decalage = i1 + LOGDIRTY_LEAF_LONG_ENTRIES;
45                 /**
46                  * Maintenant on recupere les adresses elles-
47                  *
48                  */
49                 l0 = ((l1 && mfn_valid(l1[decalage])) ?
50                     map_domain_page(l1[decalage]) : NULL);
51                 if(l0)
52                 {
53                     if(l0[i0]!=0)
54                     {
55                         count++;
56                         printk( "%lx\n", l0[i0]);
57                     }
58                     unmap_domain_page(l0);

```





```

58         }
59     }
60
61     }
62 }
63
64     ....
65
66 }
67 printk( "[END_WSS]\n" );
68 printk( "%d\n", count );
69
70     ....
71 }
```

Code 4.9: Portion de code modifiée dans la méthode `paging_log_dirty_op`

## Annexe 6 : Version 0 de l'algorithme d'estimation du WSS

```

1  #!/bin/bash
2
3  ### Params :
4  ## $1 : id de la VM
5  ## $2 : temps d'observation
6
7  ### Example :
8  ## ./script_collect.sh 2 30
9  ## Ceci collecte les logs de la VM 2 pendant 30 mins
10
11 i=0
12 sudo xl clean-dirty-bitmap $1
13 xl dmesg -c
14 period=$((SECONDS + 60*$2))
15 while(("$SECONDS" <= "$period"))
```





```
16 do
17     sleep $3
18     xl dmesg -c
19     xl collect-dirty-logs $1
20     xl dmesg -c > log/log$i
21     ((i++))
22 done
23
24 sleep 10
25 ./convert.sh 0 $i
26 cd Scriptsplots/
27 python globalTreat.py $((i-1))
28 sh Scriptsplots/globalPlot.sh
```

Code 4.10: Script de collecte des logs

