View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

SURVEILLANCE DE L'EXÉCUTION ET ANALYSE DE PRÉEMPTION ENTRE MACHINES VIRTUELLES

MOHAMAD GEBAI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUIN 2014

© Mohamad Gebai, 2014.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

SURVEILLANCE DE L'EXÉCUTION ET ANALYSE DE PRÉEMPTION ENTRE MACHINES VIRTUELLES

présenté par : GEBAI Mohamad
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme BOUCHENEB Hanifa, Doctorat, présidente
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. KHOMH Foutse, Ph.D., membre

En mémoire de mon grand-père,
A. Tehini,
Décédé le jour de ma graduation.
Que son âme repose en paix.

In memory of my beloved grandfather,
A. Tehini,
Who passed away on my graduation day.
May his soul rest in peace.

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche, le Professeur Michel Dagenais, un mentor dont la passion et l'expertise nous sont d'une grande inspiration. L'aide et le soutien qu'il m'a offert m'ont été plus qu'essentiels pour mener à bien ce travail. Je voudrais ensuite remercier la Professeure Hanifa Boucheneb qui m'a présenté au monde des systèmes d'exploitation en m'offrant plusieurs opportunités lors de mes études du premier cycle.

Je remercie tous mes collègues du laboratoire, particulièrement Francis pour les nombreuses discussions que nous avons eues et pour ses valeureux conseils, Julien pour l'orientation qu'il m'a offerte lors du début de mon projet, et Yannick pour son partage de connaissances. Un merci particulier à Geneviève pour son aide précieuse et indispensable.

Je remercie mes parents, Youssef et Leila, qui m'ont toujours poussé à entamer des études supérieures et qui ont sacrifié énormément au cours des dernières années. Je remercie mes frères Ahmad, Ali et Adam ainsi que mes amis pour leur support moral durant mes études. Merci à Ghassan de m'avoir tenu compagnie durant les nuits blanches qui ont permis à ce mémoire de voir le jour.

Je tiens finalement à reconnaître le soutien financier offert par Ericsson et le projet de recherche CRSNG qui a permis à cette étude d'être entreprise.

Ce mémoire a été écrit selon la graphie traditionnelle et révisé par l'outil de rédaction Antidote, auquel j'ai contribué.

RÉSUMÉ

Au cours des dernières années, l'infonuagique a pris un essor considérable dans le domaine de l'informatique. Cette technologie oriente les parcs informatiques à grande échelle vers un modèle flexible, portable et souvent écoénergétique. Les fournisseurs de services infonuagiques offrent à leurs clients un environnement isolé, disponible, et surtout adapté à leurs besoins. Les machines virtuelles non seulement répondent à ces critères, mais procurent aussi des avantages supplémentaires. Les coûts de gestion de matériel, de consommation d'électricité et de licences de logiciels sont réduits dans un tel modèle. Ainsi, dans un nuage informatique, un grand nombre de machines virtuelles (VM, ou système invité), gérées par un hyperviseur, cohabitent sur un même poste physique dont elles se partagent les ressources. Cependant, les machines virtuelles sont entièrement isolées les unes des autres et ont toutes l'illusion d'avoir un accès exclusif et absolu aux ressources de leur hôte. Ces ressources sont souvent surengagées, dans le sens où elles apparaissent comme étant plus disponibles qu'elles ne sont réellement. Cette caractéristique introduit inévitablement des baisses de performance aux applications des machines virtuelles, dont les sources dépendent de l'environnement externe et sont à priori impossibles à identifier. L'objectif de cette étude est d'offrir aux administrateurs des machines hôtes un outil permettant de surveiller les interactions des systèmes d'exploitation invités et localiser facilement les anomalies de performance lorsqu'elles surviennent.

Notre approche est basée sur le traçage, une méthode de surveillance qui s'est avérée efficace pour la compréhension du détail de fonctionnement d'une application. Par ailleurs, le traçage noyau permet d'effectuer la surveillance de vue globale d'un système d'exploitation. On peut ensuite analyser les interactions entre les applications utilisateurs qu'il sert et le matériel disponible qu'il exploite. En traçant simultanément le système hôte et les machines virtuelles, il est possible d'agréger les informations collectées de ces différentes sources et de présenter en clair à l'utilisateur les interactions entre les différents systèmes qui sont à priori invisibles. Cette étude se concentre sur les CPUs comme ressources partagées. Le déni du CPU à une machine virtuelle par le système hôte y introduit des délais invisibles, mais bien présents.

Nous utilisons LTTng, un traceur à bas surcoût d'utilisation, pour collecter des traces noyau générées sur les différents systèmes d'exploitation. Une fois les traces agrégées, il est possible de modéliser l'ensemble des systèmes concernés et d'effectuer un bilan d'utilisation et d'interactions entre ceux-ci. L'indisponibilité du CPU pour chaque VM est alors présentée

à l'administrateur du parc informatique, qui peut par la suite prendre les mesures nécessaires pour remédier aux problèmes. Cependant, tel que nous l'expliquerons dans ce mémoire, l'agrégation des traces est une tâche non triviale. En effet, chaque système d'exploitation, hôte ou invité, est responsable de sa propre gestion d'horloge. Par conséquent, les estampilles de temps assignées aux évènements des traces ne sont cohérentes que dans le contexte propre à chaque système. Un algorithme de synchronisation de traces distribuées a été adapté au contexte des machines virtuelles pour remédier à la non-conformité des horloges.

ABSTRACT

Over the past few years, Cloud computing has enjoyed a considerable growth in the field of computer science. This technology is leading warehouse-scale data clusters towards a more flexible, portable, and environmentally friendly model. Cloud providers offer to their customers environments that are isolated, available, and above all adapted to their needs. System virtualization not only meets these criterias, but also provides additional benefits. Management costs of hardware, energy consumption and software licenses are reduced in such a model. Thus, in a Cloud environment, a large number of virtual machines (VM, or guest system), managed by a hypervisor, coexist on the same physical computer whose resources are shared amongst them. However, virtual machines are fully isolated from one another and all have the illusion of absolute and exclusive access to the host's resources. These resources are often overcommitted whereas they appear as being more available than they actually are. This property inevitably induces performance setbacks in virtual machines. Because of system isolation, it is a priori impossible to find the source of performance problems that depend on the external environment. The objective of this study is to provide the host administrators with a tool to monitor the interactions of the guest operating systems, allowing them to easily locate and justify performance anomalies when they occur.

Our approach is based on tracing, a monitoring method that has been proven effective to give detailed information about the functioning of an application. Furthermore, kernel tracing allows to monitor the operating system and assess the interactions between users' applications it serves and the available material which it operates. By tracing the host and the virtual machines simultaneously, it is possible to aggregate the information collected from these various systems and present clearly to the user the interactions between the various systems that are usually invisible. This study focuses on the CPU as a shared resource. The denial of a CPU by the host to a VM introduces invisible yet effective latency.

We use LTTng, a low overhead userspace and kernel tracer, to collect kernel traces generated on different operating systems. Once the traces are aggregated, it is possible to model all of the systems concerned and to summarize resources contention and interactions between them. The unavailability of the CPU for each VM is then presented to the administrator of the host, which can then take the necessary steps to remedy the problems. However, as we present in this thesis, aggregation of traces turned out to be a non-trivial task. Indeed, each operating system, host or guest, is responsible for its own timekeeping. Therefore, drifts

between the clocks of the different systems are introduced, and the timestamps assigned to trace events are not consistent across traces. A portable method to address clock drifts and trace synchronization is also proposed.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AMD	Advanced Micro Devices (marque de commerce)
APIC	Advanced Programmable Interrupt Controller
ASID	Address Space Identifier
AVD	Android Virtual Device
BT	Binary translation
CLI	Clear Interrupts
CLR	Common Language Runtime
COW	Copy-On-Write
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CR3	Control Register 3
CTF	Common Trace Format
EPT	Extended Page Table
ES	Entrées/Sorties
FSM	Finite State Machine
GPA	Guest Physical Address
GVA	Guest Virtual Address
HPA	Host Physical Address
HVA	Host Virtual Address
IF	Interrupt Flag
IO	Input/Output
IP	Instruction Pointer
IRQ	Interrupt Request
ISA	Instruction Set Architecture
JIT	Just In Time
JVM	Java Virtual Machine
KSM	Kernel Samepage Merging
KVM	Kernel-based Virtual Machine
LRU	Least Recently Used
LTTng	Linux Trace Toolkit - next generation
LXC	Linux Containers
MMU	Memory Management Unit
MOM	Memory Overcommitment Manager

NPT	Nested Page Table
NUMA	Non Uniform Memory Access
OS	Operating System
PD	Page Directory
PDP	Page Directory Pointer
PID	Process Identifier
PIT	Programmable Interrupt Timer
PML4	Page Map Level 4
PT	Page Table
RAM	Random-Access Memory
RCU	Read-Copy-Update
RSS	Resident Set Size
RVI	Rapid Virtualization Index
TCP	Transmission Control Protocol
TID	Thread Identifier
TLB	Translation Lookaside Buffer
TMF	Tracing and Monitoring Framework
TSC	TimeStamp Counter
UST	UserSpace Tracing
VMCS	Virtual Machine Control Strucutre
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMX	Virtual Machine eXtension
VDSO	Virtual Dynamically linked Shared Object
VPID	Virtual Processor Identifier
VSA	Virtualization Scheduling Analyzer
VT	Virtualisation Technology
vTLB	virtual TLB

CHAPITRE 1

INTRODUCTION

Les systèmes distribués et parallèles ont pris de plus en plus d'ampleur dans le monde de l'informatique au cours des dernières années. Les parcs informatiques sont munis de plus en plus de noeuds, les noeuds de plus en plus de coeurs, et les coeurs de plus en plus de fils d'exécution. Le progrès matériel influence de près le développement logiciel, et la notion de parallélisme dans les applications modernes est omniprésente. Cependant, tous les problèmes de performance ne peuvent nécessairement être réglés en ajoutant des unités de calcul ou en répartissant les tâches. C'est par exemple le cas lorsque les ressources du système ne suffisent pas aux demandes des applications. De plus, le parallélisme a mis en place une infrastructure difficile à examiner lorsque des délais et anomalies d'exécution surgissent. Les progrès matériels et logiciels ont donc inévitablement créé un besoin essentiel pour des outils d'investigation à la hauteur de l'avancée technologique.

Dans ce travail, nous nous intéressons aux problèmes de performance liés aux anomalies d'exécution spécifiquement dans le contexte des machines virtuelles, un élément clé de l'infonuagique (Cloud Computing) qui se voit actuellement accorder un intérêt considérable. Notre étude est basée sur le traçage à un très bas niveau, à savoir le noyau du système d'exploitation. Le traçage s'est avéré une méthode efficace pour examiner le comportement d'une application ou d'un système. Cependant, une trace noyau contient généralement une quantité énorme d'information rendant son interprétation ardue, sinon impossible, pour un humain. Dans ce qui suit, nous proposons une approche permettant d'interpréter facilement et efficacement les traces noyau générées. L'objectif d'une telle analyse est de pouvoir examiner les interactions entre les machines virtuelles à travers les ressources partagées, et plus spécifiquement le CPU, ainsi que les répercussions que ce partage peut avoir sur les applications. Avant de présenter le cœur du travail, nous commençons par introduire quelques définitions de base ainsi que les éléments clés de la problématique étudiée.

1.1 Définitions et concepts de base

Cette section introduit quelques concepts de base requis pour la compréhension du travail.

1.1.1 Introduction au traçage

Le traçage est une forme de journalisation permettant d'obtenir un bilan de l'exécution d'une application ou d'un système. La forme la plus primitive de traçage est l'utilisation de clauses de type *printf*, souvent utilisées dans des applications simples par le programmeur à l'étape de développement pour avoir un compte rendu rapide de l'exécution de son application. Cette méthode est souvent utilisée pour vérifier un branchement, une entrée à une fonction, la valeur d'une certaine variable, ou quelqu'autre évènement jugé d'importance au cours de l'exécution. Ce type de traçage n'est généralement pas viable ; le code devient moins lisible, la sortie du programme est polluée, le format des traces n'est pas normalisé, et ce genre d'Entrée/Sortie (ES) apporte généralement un surcoût significatif à l'application, sans compter que l'utilisateur n'a parfois pas directement ou facilement accès à la sortie du programme. Dans cette étude, nous utilisons un outil dont la fonctionnalité et de pouvoir générer des traces de façon plus sophistiquée, notamment en utilisant des points de traces au lieu des clauses *printf*.

1.1.2 Point de trace

Pour pouvoir être tracée, une application (ou le système d'exploitation lui-même) doit être instrumentée avec des points de traces. Tout comme l'utilisation de clauses *printf* introduite précédemment, les points de trace sont écrits directement dans le code source à des emplacements stratégiques. Le but d'une telle instrumentation est de potentiellement présenter de l'information utile à l'utilisateur dans le but d'avoir une idée détaillée de l'exécution du programme analysé (au *runtime*).

1.1.3 Évènement

Un évènement est un fait marquant de l'exécution d'une application. Il possède la caractéristique d'être ponctuel, c'est-à-dire qu'il n'a aucune durée dans le temps, mais indique simplement une occurrence. Lorsqu'une trace est générée, chacune de ses entrées est un évènement de l'exécution. Un évènement a la caractéristique d'être situé dans le temps par une estampille de temps, et par conséquent les informations qu'il rapporte reflètent l'état du système au moment précis de son enregistrement. Par exemple, un changement de contexte à un certain moment sur un processeur est une information cruciale pour pouvoir suivre le flot d'exécution d'un système, son instrumentation est alors justifiée. Une trace est donc une agrégation d'évènements qui ont été préalablement instrumentés avec des points de trace.

1.1.4 Traçage noyau

Le traçage est réalisé dans le noyau Linux à l'aide de la macro `TRACE_EVENT` [1]. Cette macro requiert plusieurs arguments, dont les plus importants sont le nom du point de trace ainsi qu'une charge utile (*payload*). Le nom d'un point de trace est généralement descriptif de l'évènement reporté et la charge utile présente un lot d'informations importantes pour la compréhension et l'interprétention de l'évènement instrumenté. Par exemple, un évènement de type `sched_switch`, qui représente un changement de contexte, contient dans sa charge utile les noms ainsi que les identifiants des processus impliqués dans le changement de contexte. En plus de la charge utile spécifique à chaque type de point de trace, il est possible d'ajouter de l'information de contexte à tous les évènements, telle que le nom du processus courant, son identificateur, ou encore la valeur des compteurs de performance matériels.

1.1.5 Arbre d'attributs

Dans le but d'expliquer comment l'analyse de traces est effectuée, nous introduisons une structure de données utile pour modéliser l'état d'un système. L'arbre d'attributs, aussi appellé *state system*, est une structure de données en arbre qui représente l'état global du système. Dans une telle structure, les feuilles de l'arbre sont des attributs auxquels des valeurs peuvent être assignées. Les noeuds de l'arbre regroupent les feuilles selon une relation logique. Par exemple, un noeud "CPU" pourrait posséder comme enfants les feuilles "Contexte", qui décrit le contexte dans lequel il opère (noyau, utilisateur, interruption, etc.), et "Thread Courant" qui contient l'identifiant du fil d'exécution qui s'exécute sur ce processeur. La figure 1.1 présente un exemple d'arbre d'attributs. Il est possible d'accéder aux attributs de l'arbre en

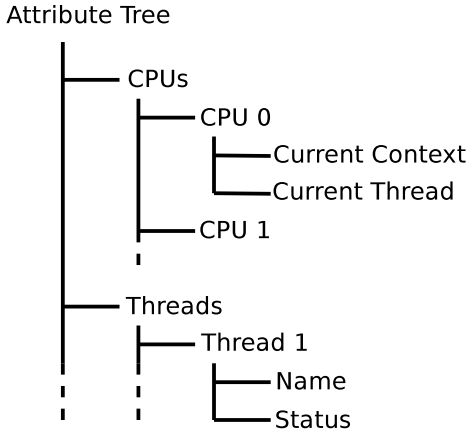

Figure 1.1 Exemple d'arbre d'attributs

utilisant un chemin d'accès similiaire à celle d'un système de fichiers. Par exemple, pour ob-

tenir l'identifiant du processus s'exécutant sur le CPU 0, il suffit de lire la valeur de l'attribut "`/CPUs/CPU0/Current Thread`".

1.1.6 Machine virtuelle

Le terme "machine virtuelle" (ou virtualisation) est généralement ambigu et peut avoir différentes significations selon le contexte dans lequel il est employé. D'un point de vue commun, une machine virtuelle est un environnement isolé dans lequel s'exécute un flot d'instructions. Dans un contexte infonuagique, une machine virtuelle (VM) désigne un "conteneur" dans lequel s'exécute un système d'exploitation (dit système invité). Cette relation d'imbrication est transparente pour le système invité qui a l'illusion d'avoir un accès direct au matériel qui lui est présenté par le conteneur. Le but de la virtualisation est de dissocier le système d'exploitation du matériel sur lequel il s'exécute. Par conséquent, il est possible d'exécuter sur une certaine architecture une application destinée à une autre. Il est également possible de "déplacer" un système d'exploitation d'un poste physique à un autre sans même l'arrêter en transférant simplement la machine virtuelle. Les fournisseurs de services infonuagiques mettent des machines virtuelles à disposition de leurs clients. Ces derniers perçoivent un accès à un système d'exploitation à part entière de façon exclusive. Le fait que ce système soit dans une machine virtuelle leur est transparent et n'altère en rien l'exécution de leurs applications.

1.2 Éléments de la problématique

Un des grands avantages de la virtualisation est l'isolement absolu des environnements invité et hôte (aussi dits respectivement *guest* et *host*). Le système d'exploitation hôte ne perçoit aucunement le flot d'exécution interne à la machine virtuelle et ne fait que lui fournir un support matériel. De la même façon, le système invité n'a aucun accès au système d'exploitation dans lequel il s'exécute. Il est ainsi possible et sécuritaire d'instancier plusieurs machines virtuelles sur un même poste physique. Cependant, les ressources physiques sont multiplexées et partagées par toutes les machines virtuelles cohabitant sur le même poste physique. Comme chacune d'entre elles est entièrement isolée du reste du système, elles ne peuvent modérer leur utilisation d'une certaine ressource pour combler une autre machine virtuelle. Notre problématique est donc de pouvoir fournir à l'administrateur de la machine physique un moyen de trouver efficacement un problème de performance dû à un partage de ressources physiques.

Le traçage s'est avéré une solution fiable et efficace pour une compréhension détaillée de l'exécution d'un système. Plusieurs travaux se basant sur le traçage de systèmes distribués ont été entamés. Dans le cas particulier des machines virtuelles, plusieurs composantes entrent en cause et font d'une telle analyse une problématique à part entière.

Tout d'abord, aucune information sur la disponibilité des ressources physique n'est partagée par les machines virtuelles. Chacune d'entre elles suppose un accès exclusif au matériel et par conséquent l'exploite autant que possible.

D'un autre côté, l'isolement des systèmes d'exploitation limite l'interprétation des traces obtenues. Habituellement, le traçage noyau est une source fiable pour effectuer un bilan de l'exécution réelle du système lui-même et des applications utilisateurs pour une certaine durée. Cependant, un système d'exploitation virtualisé tracé ne reflète pas nécessairement le flot d'exécution réel et global, puisque les contraintes imposées par la virtualisation ne sont pas, ni ne peuvent être, prises en compte par le traceur. Par conséquent, la trace enregistrée ne reporte que la perception de la machine virtuelle des évènements, biaisée par la couche de virtualisation. Notre problématique doit donc prendre en compte cet aspect et agréger les traces obtenues des différents systèmes d'exploitation impliqués pour une certaine durée.

Finalement, l'union des traces est une tâche qui s'est avérée non triviale. La dissociation des systèmes d'exploitation lègue la gestion du temps à chacun d'eux. En conséquence, les horloges de chacun des systèmes sont décalées les unes par rapport aux autres. Les estampilles de temps des évènements de chacune des traces n'ont pas la même référence de temps, et leur agrégation dans ce cas ne se limite pas à unir et trier les évènements ; le décalage d'horloges doit être pris en compte et compensé.

1.3 Objectifs de recherche

Nous nous proposons de répondre à la question de recherche suivante :

> Est-il possible de développer un modèle flexible permettant d'investiguer facilement les sources de baisse de performance liée au surengagement du CPU dans les machines virtuelles en utilisant le traçage noyau ?

Pour y parvenir, nous avons formalisé nos objectifs de recherche de la façon suivante :

1. Établir des règles permettant de modéliser l'état global du système en prenant en compte les contraintes imposées par la couche de virtualisation ;
2. Traduire la trace qui est une série d'évènements en un modèle de haut niveau selon les règles de modélisation établies ;
3. Valider que le modèle obtenu permette de détecter les sources de baisse de performance et les anomalies liées au CPU dans une machine virtuelle ;
4. Proposer une solution pour la synchronisation des évènements ;
5. Valider la méthode utilisée pour la synchronisation des évènements ;
6. Implémenter le modèle et fournir un outil utilisable ;

1.4 Plan du mémoire

Le mémoire est structuré comme suit : le chapitre 2 se propose d'établir une revue de littérature en présentant un état de l'art sur le traçage, la virtualisation, ainsi que les techniques de surveillance de machines virtuelles. Le chapitre 3 présente notre méthodologie pour répondre à la problématique établie précédemment. Le chapitre 4 fait référence à l'article de journal “Fine-Grained Preemption Analysis Across Virtual Machines”. Le chapitre 5 établit une discussion générale des résultats obtenus ainsi que des possibilités d'avancement du projet. Le chapitre 6 conclut ce mémoire.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Traçage noyau et monitorage

2.1.1 LTTng

Dans cette étude, nous utilisons LTTng comme traceur noyau, un outil dont l'objectif principal est le bas surcoût tel que présenté par Desnoyers et Dagenais (2009) [2]. Depuis la version 2.0, LTTng prend la forme de modules noyau qui doivent être chargés en mémoire lors du traçage. Par ailleurs, des composantes utilisateurs sont requises pour utiliser ces modules, interagir avec le noyau et générer des traces. Pour réduire le surcoût du traçage, LTTng utilise des tampons propres à chacun des CPUs au lieu d'utiliser un tampon partagé protégé par des mécanismes de synchronisation. Par ailleurs, des mécanismes de synchronisation RCU (Read-Copy-Update) sans verrou sont utilisés lors d'accès concurrents à des ressources partagées, tel qu'expliqué dans [3].

Lors de l'exécution d'une application instrumentée, il est possible d'activer et de désactiver le traçage au besoin. Lors de la fin de la session de traçage, LTTng produit sur disque une trace dans le format CTF (Common Trace Format) [4]. Il est ensuite possible d'analyser le résultat par conversion vers un format texte lisible avec un outil comme babeltrace, ou encore d'utiliser un outil d'analyse de traces sophistiqué comme TMF. LTTng possède également la capacité de tracer des applications en espace utilisateur (UST - UserSpace Tracing). Dans les versions les plus récentes de ce traceur, il est possible de transmettre une trace sur le réseau vers un poste dédié à l'analyse, au fur et à mesure de son enregistrement (*live streaming*).

2.1.2 Ftrace

Ftrace, pour *Function Tracer*, est un traceur sous Linux exclusivement en espace noyau. Il peut être contrôlé par l'utilisateur via le pseudo système de fichiers *debugfs*. Divers modules basés sur *ftrace* ont été développés, et permettent à l'utilisateur d'appliquer des profils d'analyses prédéfinis. Par exemple, la configuration `sched_switch` permet de tracer les changements de contexte alors que `wakeup` permet une analyse du temps entre le réveil d'une tâche et sa sélection pour exécution par l'ordonnanceur (*wake-to-schedule-in time*). De plus, des variantes de ce traceur ont été introduites. Par exemple, *function_graph* fournit une représentation graphique des entrées et sorties aux fonctions du noyau, tel un profileur. Des analyses

peuvent ensuite être entamées pour obtenir la durée d'exécution de chaque fonction. Ftrace supporte également les points de trace de la macro `TRACE_EVENT` et utilise par défaut le registre TSC (sur x86) pour assigner les estampilles de temps aux évènements qu'il enregistre. Par défaut, les traces sont enregistrées en format texte lisible par un humain. Cependant, il est possible de configurer ftrace pour une écriture plus compacte des traces en format binaire et utiliser un outil de visualisation pour en faire la lecture.

2.1.3 Perf

Un autre outil d'analyse de performance majeur sous Linux est Perf [5]. Intégré au noyau depuis la version 2.6.31, ce traceur et profileur permet entre autres de générer des traces noyau ainsi que de fournir des statistiques sur le comportement d'une application, notamment en rapportant les valeurs de compteurs de performance matériels tels que les fautes de cache et de TLB. Une composante spécifique à KVM a été introduite dans Perf dans le but d'étendre son domaine d'analyse aux machines virtuelles, ainsi que de présenter de l'information spécifique à celles-ci. Perf kvm [6] permet entre autres de donner le nombre de vmexit, ainsi que leur raison en temps réel ou dans un fichier sur disque. À titre d'exemple, nous fournissons le résultat de la commande "`perf kvm stat live`" qui permet de rapporter des statistiques sur une machine virtuelle en temps réel :

```
Analyze events for all VMs, all VCPUs:

             VM-EXIT  Samples Samples%   Time%  Min Time  Max Time         Avg time

EXTERNAL_INTERRUPT      499   39.92%   0.15%       1us      34us      3.23us ( +-  2.70% )
            VMCALL      486   38.88%   0.03%       0us       1us      0.67us ( +-  0.93% )
       APIC_ACCESS      259   20.72%   0.09%       1us       8us      3.93us ( +-  1.30% )
               HLT        4    0.32%  99.73%   11973us  500256us 272102.30us ( +- 48.64% )
     EXCEPTION_NMI        2    0.16%   0.00%       0us       1us      0.90us ( +- 16.00% )

Total Samples:1250, Total events handled time:1091364.12us.
```

Nous constatons dans cet exemple que près de 80% des évènements `vmexit` ont été causés par des interruptions de l'hyperviseur (ligne External interrupts) et par des hypercall (ligne VMCall). Nous constatons aussi que, pour la durée de l'analyse qui est un peu plus d'une seconde, 259 `vmexit` dus à des accès au registre APIC ont été effectués, ce qui est explicable par un noyau Linux invité paramétré à une fréquence d'interruption de 250 Hz (voir le paramètre `CONFIG_HZ` du noyau Linux). Les plafond et plancher des temps passés dans chacun des évènements `vmexit` sont aussi rapportés par l'outil.

2.1.4 Top

Un des outils d'analyse les plus simples d'utilisation sous Linux est top. Bien que cet outil ne donne en général que les taux d'utilisation du CPU et de la mémoire des processus s'exécutant sur le système, une composante de paravirtualisation y a été introduite. Lorsqu'invoqué dans une machine virtuelle exécutant un noyau Linux supportant la paravirtualisation, top présente une colonne additionnelle, *steal time* abrégé "st", qui donne la durée pendant laquelle un des CPUs de la machine virtuelle est préempté sur le CPU physique sur lequel il s'exécute. Cette valeur est non nulle uniquement si le CPU virtuel de la machine est préempté alors qu'il n'est pas "au repos". Le calcul de telles statistiques est possible par l'entremise d'une coopération entre KVM et le noyau Linux virtualisé.

2.2 Lecture et analyse de traces

2.2.1 Babeltrace

Babeltrace permet de convertir une trace générée en format CTF vers un format texte lisible par un humain. Bien que cet outil soit simple d'utilisation et facilement scriptable pour le calcul de métriques simplistes, sa capacité d'analyse est limitée, et se résume à la lecture de traces. Dans cet extrait simplifié de la commande *babeltrace* sur une trace générée a priori, le format est le suivant :

```
[estampille_de_temps]  nom_de_l'évènement: { numéro_du_cpu } ,
    { éléments_du_contexte } ,
    { charge_utile }
```

```
/home/mogeb/traces $> babeltrace trace_exemple/
[12:21:51.565643330]  sched_wakeup: { cpu_id = 2 } ,
    { pid = 0, tid = 0, procname = "swapper/2" } ,
    { comm = "Chrome_SyncThre", tid = 3574, prio = 120, success = 1, target_cpu
    = 2 }
[12:21:51.565644722]  sched_switch: { cpu_id = 2 } ,
    { pid = 0, tid = 0, procname = "swapper/2" } ,
    { prev_comm = "swapper/2", prev_tid = 0, prev_prio = 20, prev_state = 0,
    next_comm = "Chrome_SyncThre", next_tid = 3574, next_prio = 20 }
[12:21:51.565650866]  sys_futex: { cpu_id = 2 } ,
```

```
    {  pid = 3535, tid = 3574, procname = "Chrome_SyncThre"  } ,
    { uaddr = 0x7F9AC9FD5898, op = 129, val = 1, utime = 0x7F9AC9FD5840, uaddr2
    = 0x0, val3 = 3574 }
[12:21:51.565652169]  exit_syscall: {  cpu_id = 2  } ,
    {  pid = 3535, tid = 3574, procname = "Chrome_SyncThre"  } ,
    { ret = 0 }
[12:21:51.565697237]  irq_handler_entry: {  cpu_id = 3  } ,
    {  pid = 0, tid = 0, procname = "swapper/3"  } ,
    { irq = 41, name = "ahci" }
[12:21:51.565700642]  softirq_raise: {  cpu_id = 3  } ,
    {  pid = 0, tid = 0, procname = "swapper/3"  } ,
    { vec = 4 }
[12:21:51.565701130]  irq_handler_exit: {  cpu_id = 3  } ,
    {  pid = 0, tid = 0, procname = "swapper/3"  } ,
    { irq = 41, ret = 1 }
[12:21:51.565701990]  softirq_entry: {  cpu_id = 3  } ,
    {  pid = 0, tid = 0, procname = "swapper/3"  } ,
    { vec = 4 }
[12:21:51.565709044]  softirq_exit: {  cpu_id = 3  } ,
    { pid = 0, tid = 0, procname = "swapper/3"  } ,
    { vec = 4 }
```

Depuis cet extrait, il est possible de recréer l'état des processus impliqués dans la trace en suivant l'ordre chronologique des évènements ainsi que leur estampille de temps. L'évènement *sched_wakeup* fait sortir le processus *Chrome_SyncThre*[*ad*] de l'état bloqué et le met dans la queue des processus prêts à être sélectionnés par l'ordonnanceur lors de sa prochaine invocation. Ensuite, le processus en question est mis en exécution lors de l'évènement *sched_switch*, soit 1392 nanosecondes plus tard. Il fait ensuite recours au système d'exploitation par le biais d'un appel système (évènement *sys_futex*) puis reste en mode noyau pour 1303 nanosecondes jusqu'à l'évènement *exit_syscall* et continue son exécution en mode utilisateur. Tous ces évènements ont été exécutés sur le processeur 2. Sur le processeur 3, le processus swapper reçoit une interruption matérielle (évènement *irq_handler_entry*) sur la ligne d'interruption 41 (champ irq dans la charge utile de l'évènement) dont le nom est ahci (champ name). Cette interruption est émise par un périphérique SATA, qui est le disque dans ce cas-ci. Lors du traitement de cette interruption qui se fait dans un mode délicat avec les interruptions

désactivées, le gestionnaire d'interruption lève une interruption de type logiciel (dite SoftIRQ - *softirq_raise*) qui représente une forme de travail différé (aussi appelé *bottom half*) à un moment plus approprié et dans un état moins sensible, lorsque les interruptions seront réactivées. Il quitte ensuite la fonction de traitement de l'interruption (*irq_handler_exit*). Les évènements *softirq_entry* et *softirq_exit* délimitent ensuite le temps d'exécution du travail reporté.

2.2.2 TMF

TMF, pour *Tracing and Monitoring Framework*, est un cadre d'applications (*framework*) destiné à l'analyse automatique de traces. Il est principalement développé par Ericsson et le laboratoire DORSAL de l'École Polytechnique de Montréal dans un cadre Libre et Ouvert. TMF est implémenté sous la forme d'un module d'extension à l'environnement de développement Eclipse. Il offre plusieurs vues graphiques générées à partir de l'analyse de traces. Sa modularité rend facile le support de plusieurs formats de traces (dont CTF) ainsi que l'ajout de diverses vues graphiques [6]. À titre d'exemple, la vue de flot de contrôle montre l'état des processus du système selon le temps ; la vue des ressources présente l'état des ressources matérielles du système (CPU, lignes d'interruption) pour la durée de la trace.

Lors de la lecture d'une trace, TMF met à jour les attributs de l'arbre d'attributs (voir section 1.1.5) selon des règles établies par le type d'analyse à effectuer sur la trace. Par ailleurs, un historique d'état est créé sur disque par TMF. Cette structure, introduite par Montplaisir et al. [7], permet d'effectuer efficacement des requêtes sur les valeurs des attributs. Ainsi, l'historique complet de l'état global du système est enregistré dans une structure de données sur laquelle il est possible d'effectuer des requêtes pour obtenir l'état du système à un instant donné.

2.2.3 VSA

Shao et al. ont présenté un modèle permettant d'analyser l'utilisation CPU d'une machine virtuelle par analyse de trace [8]. Leur outil, nommé VSA (Virtualization Scheduling Analyzer), fournit des métriques intéressantes sur l'utilisation CPU et la préemption entre les machines virtuelles. Dans leurs travaux, Xen est utilisé comme hyperviseur, et Xentrace est le traceur employé. Leur modèle n'est donc pas basé sur le traçage noyau, mais bien sur le traçage de l'hyperviseur qui est une approche développée sur mesure. L'objectif de VSA est d'analyser l'ordonnanceur de Xen. Tel qu'expliqué dans la section 2.3.3, Xen est un hyperviseur de type 2, c'est-à-dire qu'il s'exécute indépendamment du système d'exploitation hôte et réimplémente certaines fonctionnalités de celui-ci, dont l'ordonnanceur. Aucune information

interne aux machines virtuelles n'est possible puisque le traçage n'est effectué qu'au niveau de l'hyperviseur, donc du côté du système hôte. Par ailleurs, pour un hyperviseur de type 1 comme KVM, l'ordonnanceur du noyau Linux est utilisé et l'analyse de ce dernier ne requiert aucun outil spécifique à la virtualisation.

2.3 Concepts de base de la virtualisation

Bien que l'intérêt porté à la virtualisation est plutôt récent dû à l'essor de l'infonuagique, l'existence de cette technologie remonte aux années 1960, lorsqu'elle fût introduite par IBM. La virtualisation offre plusieurs avantages pour les parcs informatiques. La dissociation du système d'exploitation de la machine physique sur laquelle il s'exécute permet une gestion plus flexible des services offerts. La cohabitation des machines virtuelles permet de réduire grandement le coût du matériel. De plus, la possibilité de migration des VMs d'un nœud à l'autre permet de conserver le système d'exploitation même lorsqu'une panne matérielle est détectée. De plus, les parcs informatiques s'adaptent en fonction du nombre de machines virtuelles actives. Autrement dit, durant les périodes de faible utilisation, les VMs sont réparties sur un plus petit nombre de machines physiques pour un modèle écoénergétique. Finalement, les contraintes de dépendances de versions entre différents services sont relâchées puisqu'il est possible d'instancier des machines virtuelles avec des versions de logiciels selon le besoin. Popek et Goldberg [9] définissent la virtualisation comme suit :

> *"A virtual machine is taken to be an **efficient, isolated duplicate** of the real machine."*

Plus formellement, la validité de la virtualisation repose sur les requis suivants :

- **Performance** : Les instructions d'une machine virtuelle doivent s'exécuter avec une performance raisonnable et une implication minimale de l'hyperviseur ;
- **Sécurité** : Une machine virtuelle doit être entièrement isolée de son environnement d'exécution ;
- **Intégrité** : Le résultat de l'exécution d'une application dans un environnement virtualisé doit être identique à celui de son exécution sur un système physique.

Dans ce qui suit, nous présentons la virtualisation, ses concepts de base, ainsi que les différentes méthodes introduites récemment qui permettent son bon fonctionnement.

2.3.1 Émulateurs et interpréteurs

L'émulation consiste essentiellement à simuler une plateforme en logiciel. Cette technique est très populaire et est souvent utilisée pour exécuter sur une certaine plateforme un code

binaire qui a été conçu pour une autre. Nous pouvons par exemple citer les émulateurs de consoles de jeux vidéos, ou encore l'outil de débogage d'applications mobiles Android, (AVD - Android Virtual Device) [10], qui consiste en un émulateur de plateforme mobile. QEMU [11] utilise aussi l'émulation logicielle pour exécuter des systèmes d'exploitation conçus pour des plateformes différentes de la plateforme hôte. Cependant, comme l'exécution se fait dans un "carré de sable" logiciel, les instructions sont interprétées par l'émulateur, ce qui ajoute un surcoût non négligeable. Lorsque la plateforme émulée est moins performante que la plateforme hôte, comme dans le cas des anciennes consoles de jeux vidéos et même des plateformes mobiles, l'émulation est une solution acceptable. Cependant, le surcoût rattaché à l'émulation de systèmes d'exploitation modernes rend cette méthode moins appropriée.

De la même façon, les interpréteurs, aussi nommés machines virtuelles de processus (Process Virtual Machine), ou encore Managed Runtime Environment, fonctionnent de façon similaire. Nous pouvons prendre l'exemple de la machine virtuelle Java (JVM) [12] dont le rôle est d'interpréter à l'exécution le *bytecode* d'un programme Java. C'est de cette manière que Java permet d'assurer la portabilité du langage. Une machine virtuelle Java est créée lors du lancement d'un exécutable, qui, tout comme un émulateur, interprète les instructions du binaire dans un "sandbox" logiciel. Il en est de même pour la machine virtuelle CLR (Common Language Runtime) de Microsoft [13] pour le langage C#. Pour plus de performance, le *bytecode* peut être compilé vers du code natif afin d'obtenir une rapidité comparable à celle d'une application native [14].

2.3.2 Conteneurs et contextualisation

Les conteneurs sont une autre forme d'instanciation de machines virtuelles (méthode aussi appelée contextualisation). Le terme "virtuel" ici est moins pertinent, dans le sens où les conteneurs n'ont aucun recours à une couche de virtualisation et leur approche consiste plutôt à l'isolation d'un système, dit contextualisé, dans un autre. Non seulement la machine virtuelle perçoit le matériel réel sur lequel elle s'exécute, mais elle utilise le même noyau du système d'exploitation hôte dans lequel elle réside. En réalité, les processus de la machine virtuelle sont simplement isolés du reste du système, mais ils restent un sous-ensemble des processus de celui-ci. Sous Linux, LXC (Linux Containers) [15] utilise la fonctionnalité des groupes de contrôle (cgroups - Control Groups) du noyau [16], introduits depuis la version 2.6.24, pour implémenter la contextualisation. Les cgroups permettent de limiter l'accès aux ressources de certains processus. Ces limites sont préservées au sein de la hiérarchie des processus, dans le sens où un processus hérite des limites fixées à son parent.

Soit E l'ensemble des processeurs auxquels a accès un processus P. Il est possible de limiter l'accès d'un enfant de P, nommé Q, à un sous-ensemble de E, nommé F. Ces li-

mites peuvent ensuite être imbriquées, c'est-à-dire que Q peut limiter l'accès de ses enfants à un sous-ensemble de F. Il en est de même pour l'accès aux différents noeuds de mémoire d'une plateforme NUMA (Non Uniform Memory Access). De plus, LXC permet l'isolation des espaces de noms (namespaces), permettant à une machine contextualisée de percevoir ses processus comme ayant des PIDs (Process Identifiers) propres à son contexte, par exemple le PID 1 est associé au processus *init*, l'ancêtre de tous les processus d'un système Linux. C'est également le cas pour la couche réseau où chaque machine contextualisée a l'illusion d'avoir une adresse qui lui est propre sur le réseau. Cette méthode justifie la qualification de LXC comme étant "chroot on steroids". Cette isolation est visible en affichant la liste des processus dans les systèmes contextualisé et hôte, tel que montré dans le tableau 2.1. D'autres outils

Tableau 2.1 Exemples démonstratifs du principe des conteneurs

Système contextualisé	Système hôte

`root@debian-lxc:~\# ps fax`
 	
`PID TTY COMMAND`
`1 ? init [3]`
`12 ? \_ /usr/sbin/sshd`
`17 tty1 \_ /sbin/getty 3840 tty1 linux`
`18 tty2 \_ /sbin/getty 3840 tty2 linux`
`19 tty3 \_ /sbin/getty 3840 tty3 linux`
`20 tty4 \_ /sbin/getty 3840 tty4 linux`
`21 console \_ /bin/login --`
`22 console \_ -bash`
	`/home/mogeb $> ps fax`
	`PID TTY COMMAND`
	`[...]`
	`4445 pts/0 lxc-start -n debian-lxc`
	`4447 ? \_ init [3]`
	`4662 ? \_ /usr/sbin/sshd`
	`4679 pts/2 \_ /sbin/getty 3840 tty1 linux`
	`4680 pts/3 \_ /sbin/getty 3840 tty2 linux`
	`4681 pts/4 \_ /sbin/getty 3840 tty3 linux`
	`4682 pts/5 \_ /sbin/getty 3840 tty4 linux`
	`4713 pts/6 \_ /bin/login --`
	`4728 pts/6 \_ -bash`
	`[...]`

`root@debian-lxc:~# uname -r` `3.12.0-custom`	`/home/mogeb $> uname -r` `3.12.0-custom`

de contextualisation plus anciens existent sous Linux, tels que VServer [17].

Les limites de cette approche qui se base sur l'isolation sont les contraintes de flexibilité qu'elle impose. En effet, puisque les processus d'une machine virtuelle s'exécutent en fait nativement dans le système hôte, mais avec de "simples" contraintes sur les ressources qu'elles accèdent, ils font donc partie de l'ensemble des processus du système hôte. En conséquence, une machine contextualisée n'exécute pas, ou ne choisit pas, son propre système d'exploitation. La flexibilité est donc sacrifiée aux dépens de la performance, qui, elle, est intouchée. Selon les cas, les conteneurs peuvent être la meilleure méthode de virtualisation offerte à un client, surtout si le lot de travail qui y est dédié repose en espace utilisateur, sans contrainte sur le noyau du système d'exploitation. La même approche est utilisée sous Solaris avec les *zones* [18], sous BSD avec les *jails* [19] et sous Windows avec *WinJail*.

2.3.3 Hyperviseurs

Un hyperviseur, aussi appelé gestionnaire de machines virtuelles (Virtual Machine Monitor – VMM), est une composante logicielle s'exécutant sur la machine hôte, permettant de gérer les machines virtuelles. Il représente la couche entre les systèmes d'exploitation virtualisés et le système hôte ou le matériel. Le rôle de l'hyperviseur est de créer et de détruire les machines virtuelles, ainsi que de leur allouer des ressources et de contrôler leur exécution. On distingue deux types d'hyperviseurs. Ceux de type 1, tels que Xen [20], VMWare ESX [21] et Hyper-V de Microsoft [22], qui sont autonomes et indépendants du système d'exploitation. Ils ont un accès direct à la couche physique qu'ils peuvent exploiter librement, selon leurs politiques et configurations, pour les machines virtuelles actives. La figure 1-a illustre cette stratégie. D'un autre côté, les hyperviseurs de type 2 sont ceux qui s'exécutent au sein d'un système d'exploitation hôte. Nous pouvons citer KVM [23] et Oracle VirtualBox [24] comme hyperviseurs de type 2, dont l'approche est illustrée dans la figure 2.1-b. Le principal rôle

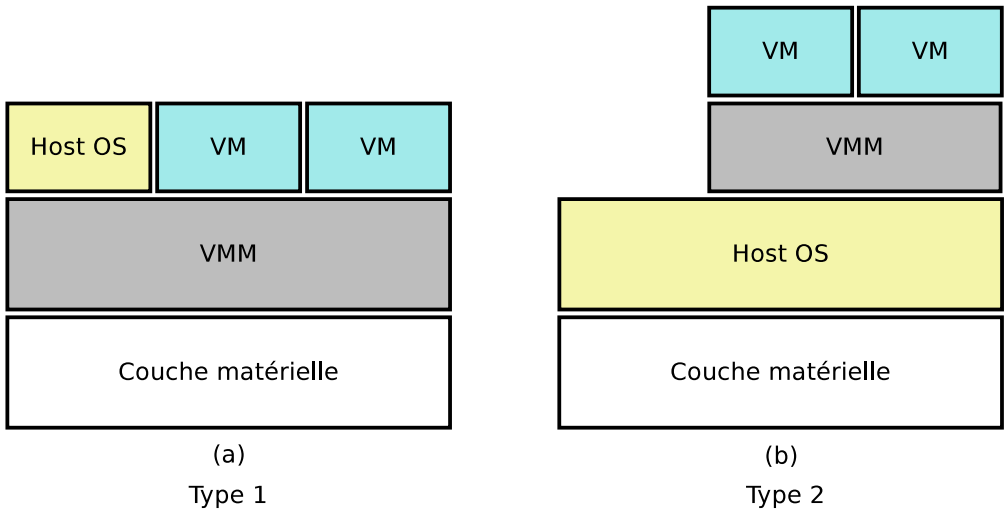

Figure 2.1 Différents types d'hyperviseurs

de l'hyperviseur est d'attribuer des ressources aux machines virtuelles qu'il assiste, ainsi que de superviser et contrôler leurs accès. Au cours de ce projet, nous utilisons KVM comme hyperviseur. Ce dernier, développé initialement par Qumranet qui fut acquise par Red Hat en 2008, fait partie du noyau Linux depuis la version 2.6.20. Il est partie intégrante du noyau, chargé sous la forme de modules noyau [25]. Deux modules sont requis, d'abord le module *kvm*, qui est indépendant du processeur et qui représente une interface plus générale à KVM, puis *kvm-intel* ou *kvm-amd* qui implémentent les fonctionnalités matérielles des processeurs Intel et AMD. Plus de détails suivront dans la section 2.4.4. Comme KVM fait partie du noyau Linux, cela réduit considérablement sa complexité. KVM peut être vu comme trans-

formant le noyau Linux lui-même en un hyperviseur, et profite donc des fonctionnalités de celui-ci, telles que les pilotes, la gestion de la mémoire, l'ordonnanceur, la gestion NUMA, et même la structure des points de traces noyau. Cela présente un grand avantage des hyperviseurs de type 2 comparativement à ceux de type 1 qui doivent implémenter eux-mêmes ces fonctionnalités.

2.4 Virtualisation du jeu d'instructions et du CPU

2.4.1 Virtualisation classique

Popek et Goldberg ont formalisé les requis d'une architecture pour pouvoir être qualifiée comme étant classiquement (ou nativement) virtualisable [9]. Typiquement, les instructions d'un jeu d'instruction (ISA – Instruction Set Architecture) d'une certaine architecture peuvent être classifiées selon les trois groupes suivants :

1. **Instructions privilégiées** : celles qui doivent être exécutées lorsque le processeur est en mode privilégié, donc par le système d'exploitation, et dont l'exécution autrement provoque une trappe.
2. **Instructions sensibles à la configuration** : celles qui tentent de modifier la configuration du système, par exemple l'écriture dans des registres de configuration. Spécifiquement pour l'architecture x86, nous pouvons mentionner les instructions `PUSHF` et `POPF` comme exemples de ce type d'instructions.
3. **Instructions sensibles à l'exécution** : celles dont le comportement dépend de la configuration du système, par exemple la lecture du niveau de privilège du CPU.

De cette classification découle le théorème permettant de qualifier une architecture comme étant nativement virtualisable :

> "*THEOREM 1. For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*"

Sachant que seul le système d'exploitation hôte s'exécute en mode privilégié, les systèmes d'exploitation virtualisés s'exécutent en mode non privilégié. Cependant, lorsque ceux-ci tentent d'exécuter des instructions sensibles, elles paraissent pour le processeur comme étant demandées en mode utilisateur. À titre d'exemple, l'instruction `POPF`, qui enregistre la valeur sur la pile dans le registre `EFLAGS`, est sensible à la configuration. Lorsqu'exécutée en mode privilégié (*protected mode*), le comportement attendu survient. En mode non privilégié, aucune action n'est prise par le processeur, aucune trappe n'est générée et le registre `EFLAGS` est non modifié. Par conséquent, le comportement est différent de celui espéré, même si c'est le

système d'exploitation (normalement privilégié) de la machine virtuelle qui a initié l'instruction. Dans le cas échéant, si le théorème avait été vérifié et donc si l'instruction sensible à la configuration faisait partie du sous-ensemble des instructions privilégiées, une trappe aurait pu être générée, ce qui aurait par la suite donné la main à l'hyperviseur pour qu'il modifie le contenu du registre `EFLAGS` au nom de l'OS invité.

Une architecture est donc classifiée classiquement virtualisable si elle respecte ce théorème et peut être virtualisée en utilisant la méthode *trap and emulate* (section 2.4.2). Les architectures destinées à usage général telles que x86 ou ARM n'ont pas été conçues en respectant ces requis, et ne sont donc pas nativement virtualisables puisqu'elles ne génèrent pas les trappes requises. Elles restent néanmoins virtualisables en utilisant des techniques différentes. La section 2.4.3 présente une alternative permettant de combler le problème, tandis que la section 2.4.4 présente les solutions proposées par les fabricants de micro-ordinateurs. Mais d'abord, nous présentons plus de détails sur la technique *trap and emulate*.

2.4.2 La méthode "Trap and emulate"

La virtualisation classique introduite précédemment se résume à une virtualisation complète en utilisant la méthode *trap and emulate*. Tel que son nom l'indique, cette technique consiste à émuler les instructions ayant causé des trappes à l'exécution sur le processeur. Lorsqu'une instruction provoque une trappe, celle-ci est interceptée par l'hyperviseur qui, à son tour, prend le contrôle de l'exécution et simule en logiciel l'instruction en question, incrémente le registre IP (Instruction Pointer) de la VM, puis redonne la main à celle-ci. Par exemple, l'instruction `CLI` (Clear Interrupts) permettant de modifier le masque IF (Instruction Flag) est privilégiée. Par conséquent, elle cause une trappe qui lègue l'exécution à l'hyperviseur qui peut modifier en logiciel le masque IF spécifique à la machine virtuelle en question.

De plus, l'hyperviseur garde en mémoire des structures de données qui décrivent en tout temps l'état de la VM à un instant donné. La lecture ou l'écriture dans un registre privilégié, tel que le registre `CR3`, cause une faute qui est interceptée par la VMM. Celle-ci consulte (ou met à jour) alors la structure de données "dans les ombres" (*shadow structures*) [26] puis retourne à la VM le contenu du registre tel qu'attendu par celle-ci.

Ceci renforce le théorème présenté plus tôt ; si toutes les instructions sensibles d'une certaine architecture sont aussi privilégiées, alors cette architecture est qualifiée comme étant classiquement virtualisable. En d'autres termes, elle peut être entièrement virtualisée en utilisant la méthode *trap and emulate*. Il est pertinent de rajouter que les transitions entre la VM et l'hyperviseur sont coûteuses. La prochaine section propose une solution différente qui est parfois préférable à *trap and emulate* puisqu'elle réduit le nombre trappes générées à

l'exécution.

2.4.3 Traduction binaire

Une alternative à la méthode *trap and emulate* est la traduction binaire (BT – binary translation), aussi appelée traduction dynamique. L'approche est similaire à la compilation *just in time* (JIT) des interpréteurs, sauf que dans ce cas-ci, seules les instructions critiques sont "traduites". L'hyperviseur remplace une instruction "sensible" (section 2.4.1) par une série d'instructions qui simulent le résultat attendu, souvent en introduisant un niveau d'indirection. Tel que présenté par [26], la traduction binaire est souvent la solution ayant une meilleure performance, même comparativement à un système non virtualisé. Ceci est explicable par le fait que les accès à certaines ressources physiques privilégiées sont traduits par de simples accès en mémoire ne requérant aucun élèvement de privilège (principe d'indirection). Dans certains cas, cette solution est préférable à la méthode *trap and emulate* puisqu'elle réduit considérablement les transitions coûteuses entre les modes du CPU. La figure 2.2 présente un exemple concret de virtualisation par traduction binaire. Les instructions de modification à des registres sensibles telles que `CLI` et l'écriture dans le registre `CR3` sont traduites par des écritures en mémoire. En utilisant la traduction binaire, l'hyperviseur VMWare arrive à virtualiser complètement l'architecture x86.

2.4.4 Virtualisation assistée par matériel

Dans le but de respecter le théorème établi par Popek et Goldberg, Intel et AMD ont tous deux introduit des extensions à l'architecture x86 la rendant nativement virtualisable. Ceci implique que l'architecture x86 pourrait être entièrement virtualisée en utilisant la méthode *trap and emulate*. Ces extensions, nommées respectivement Intel-VT (Virtualization Technology) [28] et AMD-V [29], ne sont pas normalisées. Par conséquent, du code spécifique à chacune d'elles doit être incorporé dans les hyperviseurs pour exploiter leurs fonctionnalités.

Ces extensions sont régies par des structures de données permettant de contrôler les instances des machines virtuelles. L'état de chacune des machines virtuelles est représenté en mémoire par cette structure de données, nommée VMCS (Virtual Machine Control Structure). Comme la VMCS est accédée directement par le matériel, la forme et le contenu de celle-ci sont spécifiés par chacun des fabricants [30], et doivent être respectés par l'hyperviseur.

L'approche est similaire pour Intel et AMD : deux modes d'exécution du CPU ont été introduits : mode invité et mode hôte. Le mode invité (*non-root mode*) exécute directement le code natif de la machine virtuelle. Le mode hôte (*root mode*) exécute le code de l'hyperviseur. La transition hôte-invité est réalisée à l'aide de l'instruction `vmrun` (`VMLAUNCH` ou

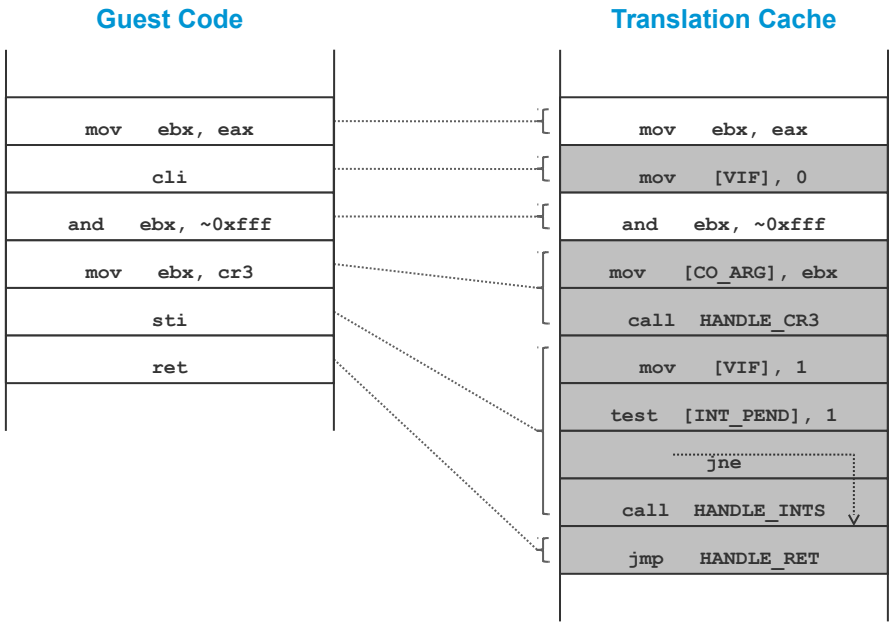

Figure 2.2 Exemple de virtualisation par traduction binaire

Source : [27]

`VMRESUME` sur Intel). Lorsqu'exécutée, cette instruction charge l'état de la machine virtuelle selon le contenu de la VMCS et donne l'exécution à la machine virtuelle. La transition inverse, invité-hôte, n'est pas réalisée à l'aide d'une instruction, mais est plutôt une réaction à une instruction privilégiée du mode invité. Lors d'un tel évènement, un champ de diagnostic permet à l'hyperviseur d'obtenir la cause de la transition. Celui-ci émule l'instruction trappée et redonne l'exécution à la VM avec l'instruction `vmcall` jusqu'à la prochaine trappe, et ainsi de suite. L'algorithme 1 donne une implémentation simplifiée exploitant ces modes d'exécution.

2.4.5 Paravirtualisation

La paravirtualisation, aussi appelée virtualisation assistée par système d'exploitation (OS-assisted virtualization) est une méthode alternative permettant d'assurer non seulement la virtualisation, mais surtout une virtualisation plus efficace. Cette approche est un compromis entre la portabilité et la performance. La paravirtualisation se définit comme étant une coopération de la part du système invité dans le but de réduire le surcoût introduit par

Algorithm 1: Algorithme de virtualisation assistée par matériel

```
start:
    load vmcs;
    vmentry;      // control is given to the VM / wait for an exit
    save vmcs;    // assume an execution trap
    read vmcs.exit_reason;
    handle exit accordingly;
    increment vmcs.IP;    // increment Instruction Pointer of the VM
    goto start;
```

la couche de virtualisation. Cette méthode requiert des modifications au noyau du système d'exploitation virtualisé. Par ailleurs, la paravirtualisation peut aussi être implémentée sous la forme de modules noyau chargés par le système d'exploitation lorsqu'il détecte une couche de virtualisation qui le sépare du matériel (section 2.6.1).

Sur les ordinateurs capables de virtualisation matérielle, l'instruction spécialisée `vmcall` permet d'initier une requête à l'hyperviseur lorsqu'exécutée par le système invité. Cette communication explicite de la VM vers l'hôte, appelée *hypercall*, est similaire à un appel système ; des arguments peuvent être spécifiés dans certains registres qui seront lus et interprétés par le système hôte pour répondre à la requête. Cependant, cette méthode impose des contraintes de portabilité et de maintenabilité. En effet, puisque la paravirtualisation est spécifique à l'hyperviseur et au matériel, le système invité doit être modifié pour coopérer avec l'hyperviseur. Sur les systèmes d'exploitation libres, ce problème est moins contraignant que sur les systèmes propriétaires qui, eux, doivent être implémentés par les développeurs du système d'exploitation. De plus, cette approche n'est pas envisageable dans certains cas, par exemple dans un contexte de test et validation d'un système d'exploitation ou d'un pilote.

2.4.6 Détection d'un environnement virtualisé

Dans la section précédente, nous avons introduit la paravirtualisation comme étant une coopération entre les systèmes d'exploitation hôte et invité. Pour ce faire, le système invité doit pouvoir détecter un environnement virtualisé à l'exécution. Cette section explique les mécanismes utilisés par le noyau Linux en se basant sur le code source de celui-ci (à la version 3.12).

Sous Linux, c'est lors de la routine d'initialisation du noyau (fonction `setup_arch()` spécifique à x86) que le noyau tente de détecter l'existence d'un hyperviseur dans lequel il pourrait être encapsulé. Tel que présenté par l'algorithme 2, un tableau de structures de type `hypervisor_x86` est défini et contient l'ensemble des hyperviseurs connus du noyau Linux qui pourraient potentiellement être détectés. Chacune de ces structures contient un pointeur

Algorithm 2: Fonction `detect_hypervisor_vendor()`

```
Input: hypervisors : list of known hypervisors

Foreach hypervisor h in hypervisors {
  if(h.detect()) {
    Hypervisor detected: h.name;
    break;
  }
}
```

vers une fonction de détection spécifique à chaque hyperviseur. Ces fonctions sont exécutées l'une après l'autre jusqu'à la détection d'un hyperviseur. Si, dans le cas échéant, aucune fonction de détection ne réussit, le noyau conclut qu'il s'exécute nativement sur la couche matérielle. Il est aussi possible qu'il s'exécute dans un hyperviseur inconnu. Si tel est le cas, alors le noyau invité ne coopère aucunement avec l'hyperviseur, ce qui aboutit à une virtualisation plus ardue et à une performance dégradée. L'algorithme 3 montre le fonctionnement de la méthode de détection de KVM. La méthode de détection se base sur la méthode *trap and emulate* introduite précédemment. Selon la documentation d'Intel, l'instruction `cpuid` cause une trappe lorsqu'exécutée en mode invité par le système d'exploitation virtualisé, ce qui donne le contrôle à l'hyperviseur. En consultant la VMCS pour obtenir la raison de la trappe, l'hyperviseur peut détecter que l'instruction `CPUID` fût exécutée par la VM. Celui-ci émule l'instruction demandée en fournissant "KVMKVMKVM" comme signature du CPU puis redonne l'exécution à la VM à l'aide de l'instruction `vmresume`. Celle-ci peut alors tester la signature du CPU et la comparer à la chaîne attendue pour chacun des hyperviseurs connus pour conclure à l'existence d'un environnement virtualisé. L'approche est similaire pour VMWare, Hyper-V et Xen qui sont connus du noyau Linux en date de sa version 3.12.

Algorithm 3: Fonction de détection de KVM `kvm_detect()`

```
unsigned int eax, ebx, ecx;
char signature[12];

cpuid(&eax, &ebx, &ecx); // inevitable trap
memcpy(signature + 0, eax, 4);
memcpy(signature + 4, ebx, 4);
memcpy(signature + 8, ecx, 4);

if(signature == "KVMKVMKVM")
  return true;
return false;
```

2.5 Virtualisation du MMU et de la mémoire

2.5.1 Concepts de base de la mémoire

La mémoire est le sous-système d'un système d'exploitation présentant le plus de difficultés à être virtualisé. Les systèmes d'exploitation modernes utilisent le mécanisme de mémoire virtuelle comme méthode d'indirection pour permettre une dissociation des espaces mémoires des processus du système. Le système d'exploitation assume la tâche de traduction d'une adresse virtuelle demandée par un processus utilisateur vers une adresse physique. Communément appelée *page walk* [31], cette tâche est réalisée en matériel à l'aide de l'unité de gestion de la mémoire (MMU) en utilisant la table de pages du processus et est généralement coûteuse. Selon l'architecture, la *page walk* résulte en plusieurs accès à la mémoire vive du système, ajoutant un surcoût non négligeable comparativement à la vitesse des CPU modernes ; un accès en mémoire requiert environ une centaine de cycles CPU. Dans le cas de la plateforme x86 64 bits, 4 accès en mémoire sont requis, suivis d'un accès additionnel pour la valeur finale dans la page, tel que le montre la figure 2.3. Ceci augmente le nombre de cycles par instruction puisque le CPU est "au repos" en attendant le retour des accès en mémoire.

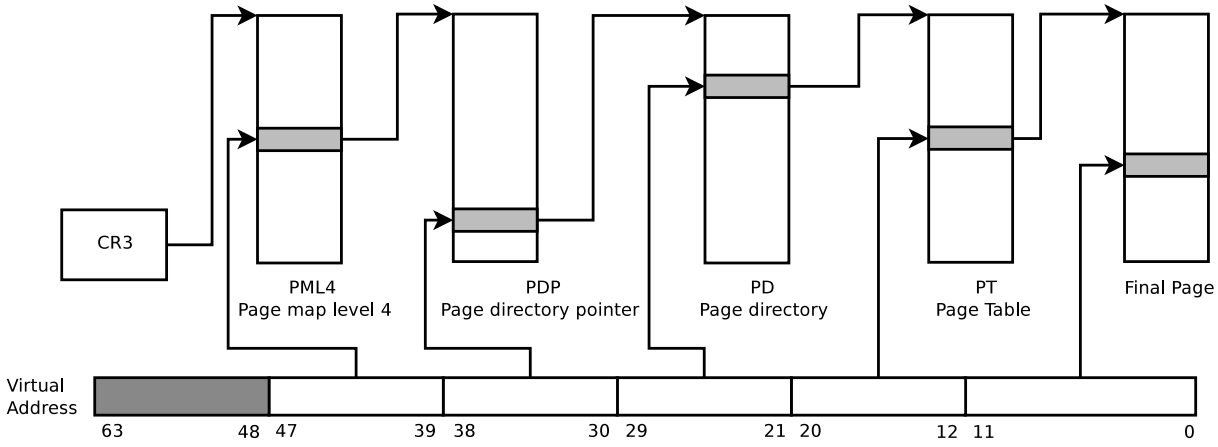

Figure 2.3 Processus de *page walk* traditionnel sur x86 64 bits

Dans le but de réduire le surcoût associé à la traduction des adresses virtuelles, les fabricants des micro-ordinateurs ont ajouté une composante matérielle au CPU, le TLB (Translation Lookaside Buffer), qui sert de mémoire cache pour la résolution d'adresses virtuelles à physiques. Comme cette information est privée et valide dans le contexte d'un unique processus, chaque changement de contexte peut requérir un nettoyage du TLB pour éviter de présenter de l'information sensible d'un processus à un autre. D'ailleurs, l'écriture dans le registre `CR3` lors d'un changement de contexte provoque inévitablement l'invalidation du TLB

(le registre `CR3` sert d'adresse de base pour la *page walk*). De plus, le système d'exploitation doit coopérer avec le processeur pour assurer une certaine cohérence du TLB. Par exemple, lorsqu'une association {*adresse virtuelle, adresse physique*} n'est plus valide dans le contexte d'un processus, le système d'exploitation doit invalider l'entrée de cette adresse virtuelle dans le TLB. Il en est de même pour les mécanismes de cohérence de cache : lors d'une modification à une table de pages partagées par plusieurs fils d'exécution sur un CPU différent, le mécanisme de cohérence de cache doit invalider l'entrée à cette adresse dans le TLB [32].

2.5.2 Besoin de virtualisation

Un système d'exploitation s'attend généralement à un accès direct au matériel auquel il est exposé. Par conséquent, les adresses physiques qu'il manipule doivent avoir une correspondance réelle au niveau du matériel. Dans le cas des systèmes virtualisés, une couche d'indirection additionnelle est présente. En effet, comme une VM est en réalité un processus régulier du point de vue du système hôte, elle doit passer par les mêmes étapes de traduction d'adresses virtuelles à physiques, en plus de la traduction dans le contexte du système virtualisé. De plus, les adresses que le système invité croit physiques sont en réalité des adresses virtuelles de l'espace mémoire du processus qui émule la machine virtuelle.

Nous présentons un exemple simple qui montre le besoin de virtualiser l'unité de gestion de la mémoire. Pour effectuer une résolution d'adresse, le contenu du registre `CR3` est utilisé comme adresse de base pour la *page walk*. Cette adresse doit forcément être physique, sinon le processus de *page walk* serait récursif et infini. Dans le cas d'un système virtualisé, le contenu du registre `CR3` contient en fait une adresse virtuelle de l'espace mémoire du processus émulant la machine virtuelle. Cette propriété rend la résolution d'adresse impossible par matériel sans mécanisme de virtualisation.

Dans ce qui suit, les termes que nous utiliserons sont ceux définis par Agesen et al. [33]. Nous distinguons 3 types d'adresses mémoire : les adresses virtuelles du *guest* (gva – guest virtual address), les adresses physiques du *guest* (gpa – guest physical address) et les adresses physiques réelles (hpa – host physical address). Le système invité est responsable de la traduction gva vers gpa, alors que l'hyperviseur (ou le système hôte) se charge de la traduction gpa vers hpa. Le reste de cette section présente les méthodes permettant une virtualisation efficace de la mémoire, en prenant en compte les contraintes de la mémoire virtuelle.

2.5.3 Shadow page table

La virtualisation de la table de pages par Shadow Page Table est une solution implémentée en logiciel pour la virtualisation de la mémoire. Cette approche se résume à une combinaison des traductions gva-gpa et gpa-hpa en une seule étape : gva-hpa. La première partie, gva-gpa, est en réalité résolue directement en utilisant la SPT, qui, tel un TLB, garde en mémoire une table de traduction des adresses gva-gpa, d'où son nom alternatif de vTLB (virtual TLB) [34]. Par la suite, la gpa est passée directement au MMU, qui se charge de résoudre la traduction gpa-hva selon le mécanisme de *page walk* habituel. Le système invité est donc responsable de ses tables de pages, mais l'hyperviseur doit en garder une copie à jour dans la SPT. Cette solution requiert donc que la SPT contienne en tout temps une traduction cohérente gva-gpa. Pour satisfaire cette condition, l'hyperviseur protège, en lecture seule, les tables de pages du système invité. Ainsi, chaque modification à une table de pages cause une trappe qui permet à l'hyperviseur d'interrompre la VM pour mettre à jour sa propre SPT et en assurer la cohérence. Cette méthode introduit un surcoût d'utilisation non négligeable dû à la fréquence des transitions entre VM et VMM, surtout sur des systèmes à virtualisation assistée par matériel où cette transition est encore plus coûteuse [26]. Tel que présenté par Bae et al. [34], des variantes à la SPT traditionnelle ont été implémentées dans le but d'optimiser son utilisation. Nous présentons ces approches pour complémentarité de l'étude, bien qu'elles ne soient pas abordées au cours de ce projet. Nous nous restreignons donc de présenter les détails de leurs implémentations.

Nous présentons d'abord la SPT par mécanisme de cache (*Shadow Paging with caching*). Dans cette alternative, les traductions gva-gpa sont gardées en mémoire même après un changement de contexte. L'hyperviseur garde donc plusieurs SPT : celle construite pour le contexte courant, et celles ayant déjà été construites pour les contextes antécédents. Cela étant dit, cette approche est plus importante en complexité ; toute modification à une table de pages doit être suivie, même si la table de pages modifiée n'appartient pas au contexte courant. Ceci est généralement atteint par protection, en lecture seule, de toutes les tables de pages du système invité.

Une approche d'optimisation alternative est la SPT par pré-lecture (*Shadow Paging with Prefetching*). Dans cette approche, lorsqu'une modification est trappée par l'hyperviseur, la totalité des traductions gva-gpa est copiée dans la SPT, contrairement au rituel de copier uniquement la traduction de l'adresse ayant causé la trappe.

2.5.4 Intel EPT et AMD RVI

La SPT est une solution acceptable lorsqu'aucune alternative matérielle n'est disponible. Cependant, sur les processeurs récents, Intel et AMD ont tous deux ajouté des extensions matérielles à leurs micro-ordinateurs pour présenter une alternative à la solution logicielle de la virtualisation du MMU. Bien qu'elles ne soient pas normalisées, ces deux extensions, nommeés Intel-EPT et AMD-RVI (autrefois NPT – Nested Page Table), sont similaires par leurs approches.

Ces extensions tentent d'éliminer le besoin de l'hyperviseur pour la traduction complète gva-hpa. Pour ce faire, un niveau additionnel de tables de pages est introduit. En effet, tel que nous l'avons expliqué dans la section 2.5.2, les adresses perçues physiques par le système invité sont en réalité virtuelles et par conséquent requièrent elles aussi une résolution vers des adresses physiques réelles. La table de pages additionnelle, dite imbriquée, permet cette résolution. La figure 2.4 met en évidence cette imbrication de tables de pages et simplifie le processus complet.

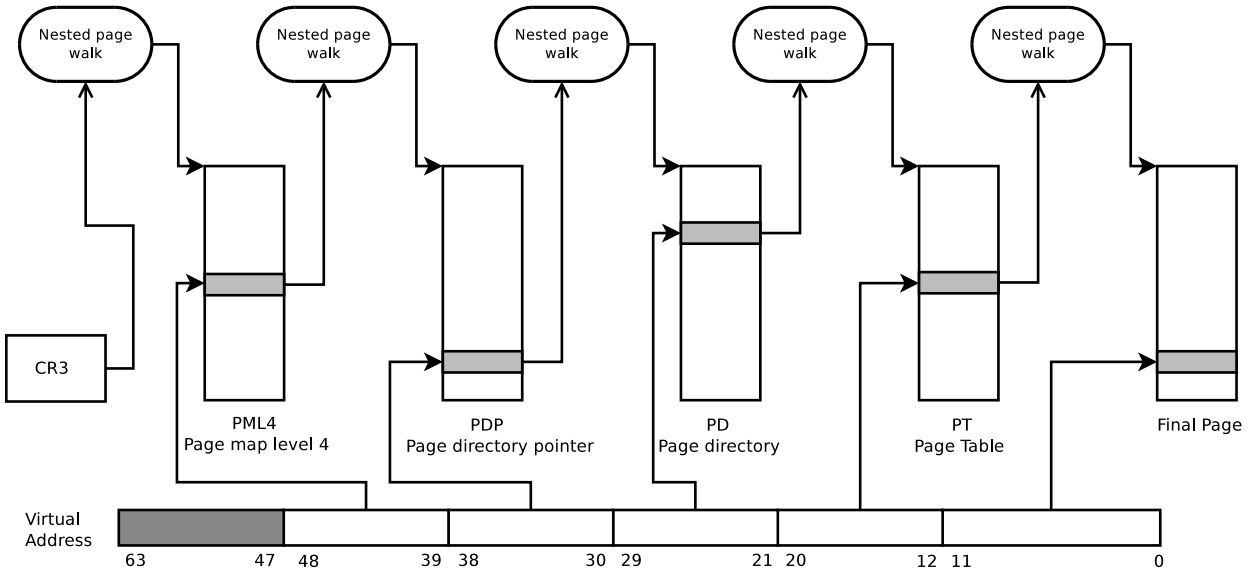

Figure 2.4 Processus de *page walk* à deux dimensions sur x86 64 bits

Ainsi, pour une résolution gva-hpa, un niveau de table de pages est dédié à chacune des étapes intermédiaires gva-gpa et gpa-hpa. La *page walk* de la table de pages du système invité permet la traduction d'adresses gva-gpa en 4 étapes (tel que mentionné précédemment). Cependant, chaque adresse gpa doit être elle-même traduite en adresse hpa par le biais d'une *page walk* dans la table de pages de l'hyperviseur. En d'autres termes, pour chacun des niveaux n de la table de pages du système invité, une *page walk* complète (en $O(n)$) de la table de pages de l'hôte est requise. Cette traduction résulte donc en un processus de

complexité $O(n^2)$, où n est le nombre de niveaux de la table de pages, d'où le nom de tables de pages imbriquées ou étendues. Ce procédé est aussi appelé *page walk* à deux dimensions (*2-dimensional page walk*). La figure 2.4 montre le processus de *page walk* à deux dimensions. Nous voyons clairement que chacune des adresses intermédiaires (PML4, PDP, PD et PT) requiert une *page walk* complète du système hôte. Le problème proposé dans la section 2.5.2 concernant le contenu du registre `CR3` pour la traduction d'adresses est résolu en utilisant la table de pages imbriquée.

Une fois la *page walk* imbriquée complétée, le résultat de la traduction gva-hpa est enregistré dans le TLB. Celui-ci peut alors contenir, selon le mode de fonctionnement du CPU, des traductions gva-hpa dans le contexte d'une machine virtuelle, ou hva-hpa dans le contexte d'un processus de l'hôte. Sur certains micro-ordinateurs, le processeur possède un TLB additionnel "imbriqué" (nested TLB) dans le but d'accélérer la *page walk* imbriquée (gpa-hpa). Selon le principe de localité des tables de pages de la VM, ce TLB additionnel élimine la *page walk* imbriquée lorsque la traduction gpa-hpa est déjà enregistrée [35].

2.5.5 TLB associatif

Grâce aux tables de pages imbriquées, la résolution d'adresses gva vers hpa ne requiert plus de transitions entre VM et VMM. Cependant, cette approche augmente considérablement le coût d'un défaut de TLB qui requiert $O(n^2)$ accès en mémoire dans de telles circonstances. Nous avons expliqué dans la section 2.5.1 que chaque changement de contexte requiert, ou plutôt provoque, une invalidation du TLB. Dans le but d'éliminer le besoin d'invalidation du TLB, les fabricants de micro-ordinateurs ont introduit des TLB associatifs. Dans cette variante du TLB traditionnel, chaque entrée se voit assigner un identificateur qui spécifie l'espace mémoire dans lequel elle est valide. Intel et AMD ont chacun leur version d'identification de l'espace mémoire, et KVM implémente la gestion des identifiants dans les modules propres à chaque type de CPU. Les implémentations diffèrent selon le type du CPU, mais le fonctionnement est globalement le même. Sur les processeurs Intel, cet identifiant est nommé VPID, *pour Virtual Processor Identifier*, alors qu'AMD utilise le terme ASID, pour *Address Space Identifier*. Des instructions spécifiques aux CPU permettent d'invalider les entrées du TLB associées à un VPID ou ASID particulier (respectivement `invvpid` et `ivlpga`).

Lors d'une *page walk*, le résultat de la résolution d'adresse virtuelle à physique est enregistré dans le TLB en combinaison avec l'ASID (ou le VPID) courant. Lors d'une future demande de résolution de la même adresse, le MMU vérifie que l'ASID courant est égal à celui assigné à une entrée du TLB. Si c'est le cas, la résolution est valide et retournée au CPU. Dans le cas échéant, une *page walk* complète est nécessaire.

L'hyperviseur est responsable d'assigner un ASID à chaque machine virtuelle et d'en effec-

tuer la gestion. KVM implémente ces fonctionnalités dans les modules *kvm_intel* et *kvm_amd* spécifiques à chaque fabricant puisque ces extensions ne sont pas normalisées. En réduisant le nombre d'invalidations du TLB, le nombre de défauts de celui-ci est également réduit, ce qui impacte grandement la performance globale du système. En effet, les transitions entre VM et VMM sont moins coûteuses, et le nombre de *page walk* à deux dimensions est globalement réduit, ce qui améliore le nombre de CPI (*Cycles Per Instruction*).

2.5.6 Surengagement de la mémoire

Définitions

Sur les serveurs de virtualisation, il est commun que la mémoire physique de la machine hôte soit "surengagée" ou, en d'autres termes, qu'elle apparaisse comme étant plus disponible qu'elle ne le soit réellement. Ainsi, la somme de la mémoire allouée aux machines virtuelles invitées est souvent supérieure à la mémoire disponible du système hôte. Cela est d'abord permis par le fait que les pages réclamées par un processus ne sont réellement allouées que lorsqu'elles sont accédées par celui-ci. Par conséquent, lorsqu'une machine virtuelle est créée, peu de pages lui sont effectivement allouées au départ bien qu'elle réclame la totalité de la RAM qui lui est configurée. Au fur et à mesure de son exécution, les premiers accès aux pages de son espace mémoire génèrent des fautes de pages mineures par le MMU, ce qui résulte en des allocations de pages physiques. De ce fait, une allocation d'une page en mémoire n'est réalisée que lorsqu'une VM, ou plutôt lorsqu'un processus de celle-ci, y accède pour la première fois. Cependant, lorsqu'un processus libère de la mémoire dans une VM, les pages de mémoire physique ne sont pas libérées par le noyau hôte. En effet, du point de vue de ce dernier, les pages restent mappées à l'espace mémoire de la VM. Aucune information de libération de mémoire n'est transmise au système hôte ou à l'hyperviseur qui leur permettrait de réduire la proportion de mémoire effectivement utilisée. Bueso et al. (2012) ont proposé une approche qui estime la taille de la mémoire requise par une VM (RSS - *Resident Set Size*) en utilisant un algorithme de *stack distance* [36]. La figure 2.5 montre un exemple d'utilisation qui prouve le besoin d'une virtualisation efficace de la mémoire. La ligne rouge suit les fautes de pages qui aboutissent à des allocations de pages mémoire sur le système hôte alors que la courbe bleue suit celles de la VM. La mémoire de 400Mb utilisée de façon ponctuelle par un processus de la machine virtuelle apparaît comme étant libérée par le système invité, bien que l'hyperviseur ne la libère jamais sur le système hôte. Elle reste donc réservée au processus QEMU qui émule la machine virtuelle.

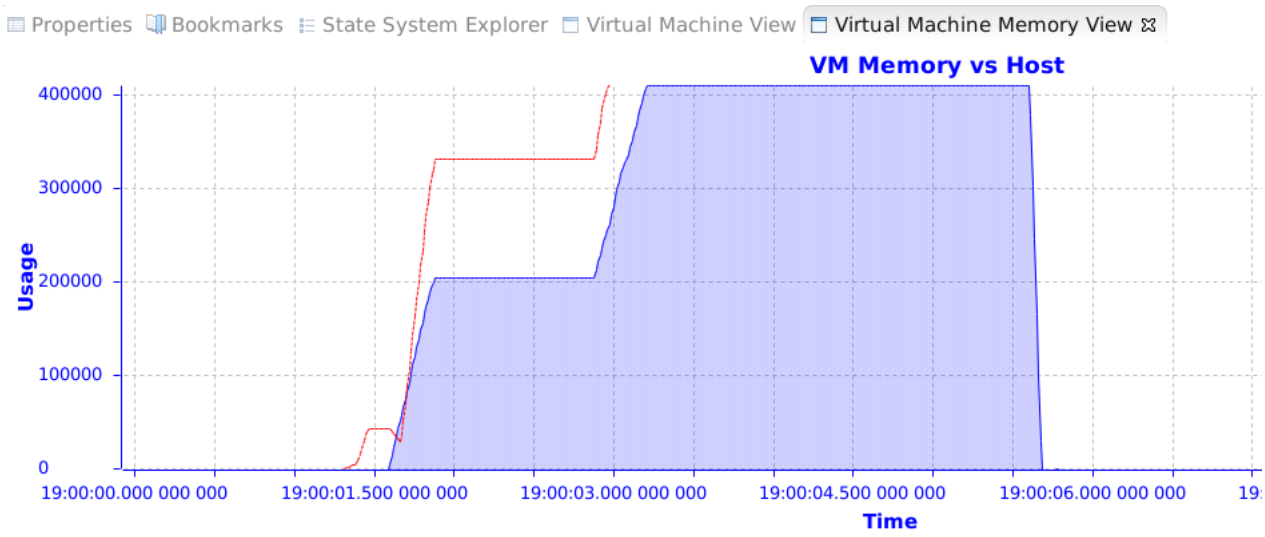

Figure 2.5 Différence entre la mémoire utilisée dans une VM et la mémoire physique qui lui est allouée ; ordonnées en kilooctets

.

Nous présentons dans cette section deux méthodes qui permettent une virtualisation efficace de la mémoire entre les machines virtuelles.

KSM

Tout d'abord, KSM [37], ou Kernel Samepage Merging, est un mécanisme permettant d'unifier plusieurs pages en mémoire propres à différents processus lorsque celles-ci ont le même contenu. Pour ce faire, un démon noyau, *ksmd*, effectue périodiquement une somme de contrôle sur les différentes pages allouées d'un système. Lorsque des pages identiques sont détectées, elles sont alors libérées et remplacées par une seule page protégée en lecture seule. Si un processus tente de modifier le contenu de cette page, celle-ci est dupliquée par le mécanisme COW (Copy-On-Write), commun sous Linux, puis modifiée uniquement dans l'espace mémoire du processus ayant initié l'écriture. KSM est configurable par l'administrateur de l'hôte, par exemple la période à laquelle le démon effectue les sommes de contrôle ainsi que le nombre de pages sur lesquelles cette opération est effectuée sont des paramètres modifiables à travers un répertoire du pseudo système de fichier *sysfs* (`/sys/kernel/mm/ksm`). KSM peut dans certains cas réduire significativement la quantité de mémoire utilisée, surtout lorsque plusieurs VM utilisent le même système d'exploitation, la même version du noyau, les mêmes librairies, etc. Pour permettre à deux pages d'être potentiellement unifiées par KSM, l'appel système `madvise()` (permettant de fournir de l'information au système d'exploita-

tion sur la gestion du bloc alloué) est requis lors de l'allocation initiale de la mémoire avec `MADV_MERGEABLE` comme paramètre, tel qu'effectué par QEMU lors de la création d'une VM.

Extraction de mémoire par *Ballooning*

Par ailleurs, il est possible à l'hyperviseur de réclamer explicitement la mémoire physique allouée à une VM. Le principe de *ballooning* permet de réduire la taille de la RAM assignée à une machine virtuelle sans l'arrêt de celle-ci, similairement à de la connexion à chaud (*memory hot-plug*). Pour y arriver, un module est chargé par le système d'exploitation virtualisé lors du démarrage. La famille virtio inclut le module *virtio-balloon* pour permettre le *ballooning*. Ce module crée un processus noyau, nommé *vballoon*, sous forme de démon (*daemon*). Sur demande de l'hyperviseur, ce processus effectue une allocation de mémoire égale à la taille que l'on veut réduire. Comme *vballoon* est un processus noyau, son espace mémoire ne peut être *swappé*. Le module *virtio-balloon* rend alors au système hôte la mémoire que le démon a allouée. Selon la demande de l'administrateur de l'hôte, le ballon gonfle et dégonfle pour retirer ou rendre de la mémoire à une machine virtuelle. La figure 2.6 schématise le processus de *ballooning*.

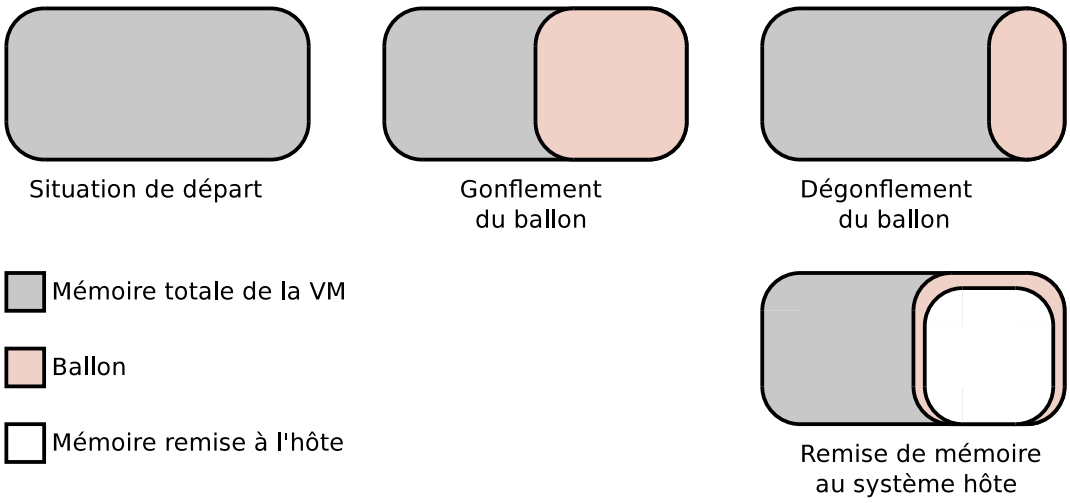

Figure 2.6 Principe du *ballooning*

L'extrait de trace suivant montre les évènements enregistrés par LTTng lors d'une demande de réduction de mémoire :

```
[01:50:24.377172537]  mm_page_alloc: {  cpu_id = 0  } ,
    {  pid = 433, tid = 433, procname = "vballoon"  } ,
    { page = 0xFFFFEA0003D229E8, order=0, gfp_flags=200922, migratetype=2 }
[01:50:24.377173693]  mm_page_alloc: {  cpu_id = 0  } ,
```

```
    {  pid = 433, tid = 433, procname = "vballoon"  },
    { page = 0xFFFFEA0003D7B250, order=0, gfp_flags=200922, migratetype=2 }
[...]
```

L'évènement `mm_page_alloc` indique une allocation d'un bloc de mémoire physique. Le champ `order` de la charge utile indique le nombre de pages allouées en puissance de 2 ; un ordre de 0 indique 2^0 page. La trace indique donc un grand nombre de pages allouées au processus *vballoon* (champ *procname*), qui sera ensuite rendu au système hôte. En gonflant puis dégonflant le ballon, le *ballooning* permet de simplement "nettoyer" les pages allouées puis libérées d'une machine virtuelle pour réduire l'écart entre le RSS d'une machine virtuelle et la quantité de mémoire physique qui lui est mappée, ce qui serait une solution au cas d'utilisation de la figure 2.5.

MOM

MOM, ou Memory Overcommitment Manager, est un composant de la pile d'outils oVirt spécifique à la gestion de la virtualisation sous Linux. Cet outil requiert des démons sur les systèmes hôte et invités qui peuvent communiquer entre eux. MOM extrait de la mémoire aux machines virtuelles selon des politiques d'utilisation préétablies par l'administrateur du système hôte (*policy-driven*). Ces règles sont typiquement définies sous forme de seuils d'utilisation de mémoire dans une machine virtuelle. Par exemple, la mémoire d'une VM se verrait réduite lorsque le taux d'utilisation sur le système hôte atteint 80% de la mémoire totale. Cependant, le choix de la machine virtuelle est généralement non optimal. En effet, MOM tente d'extraire la mémoire des machines virtuelles les plus gourmandes. Cependant, celles-ci pourraient potentiellement se partager un grand nombre de pages mémoire à travers KSM. Dans un tel scénario, le travail de MOM est plutôt nuisible puisque la quantité de mémoire effectivement libérée serait inférieure à celle demandée, et ce, à cause des fautes de pages qui créent des copies par COW. D'un autre côté, il est possible qu'une machine virtuelle ait un RSS négligeable, mais qu'elle ait auparavant touché une grande plage mémoire. Comme nous l'avons expliqué, ceci garde la mémoire physique mappée à la machine virtuelle, bien qu'elle ne lui soit plus utile. Dans ce cas-ci, cette machine virtuelle ne verrait jamais sa mémoire réduite selon les règles de MOM.

2.5.7 Virtualisation de la page cache

Définitions

La *page cache*, aussi appelée *disk cache*, est une fonctionnalité commune des systèmes d'exploitation modernes [38] qui se base sur le principe de localité. Elle représente une cache

en mémoire pour les secteurs d'un *block device* (typiquement un disque sur les ordinateurs personnels). Sous Linux, tous les accès aux mémoires de masse se font à travers la mémoire principale du système. Ainsi, lors de la lecture d'une valeur sur disque, le secteur contenant cette valeur est supposé accédé prochainement, et est entièrement copié en mémoire pour optimiser les références futures.

De la même façon, les écritures à des valeurs sur disque se font d'abord en mémoire, et leur propagation au *block device* est différée à un moment ultérieur pour une écriture asynchrone, sauf si le contraire est explicitement spécifié (voir l'option `O_SYNC` de `write()` pour une écriture synchrone). Une écriture synchrone affecte la performance de l'application qui doit bloquer en attendant la propagation des données vers le disque avant de reprendre son exécution. Elle empêche aussi d'autres optimisations potentielles, telles que le réordonnancement ou l'union des requêtes par le système d'exploitation. La *page cache* est donc la mémoire physique non allouée du système qui est utilisée pour servir de cache entre le CPU et le disque (aussi classifiée comme mémoire libérable – *freeable memory*).

Paramétrisation de la page cache

Par ailleurs, certaines configurations permettent d'encourager l'utilisation de la *page cache*, comme le paramètre `swappiness`. Ce dernier permet de contrôler l'agressivité du système d'exploitation à déplacer les pages les moins récemment utilisées de la mémoire vers un périphérique de stockage externe (typiquement un disque). Le noyau Linux maintient en mémoire des listes LRU (Least Recently Used) qui gardent un compte des accès aux pages allouées. Selon la valeur de `swappiness`, les pages les moins nouvellement référencées peuvent être sujettes au *swapping*.

Le paramètre `swappiness` est une valeur entière entre 0 et 100 qui peut être modifiée via le pseudo système de fichiers *procfs*. Une valeur élevée est une indication au système d'exploitation d'une plus grande agressivité pour *swapper* les pages les moins récemment utilisées. Ce concept peut paraître contre-intuitif puisque le *swapping* vers une mémoire de masse impacte fortement la performance et est généralement à éviter. L'intérêt de *swapper* des pages volontairement permet de libérer de la mémoire qui pourrait être utilisée comme *page cache* et accélérer les futurs accès au disque. La valeur par défaut du paramètre `swappiness` est de 60 sous la plupart des distributions Linux.

Virtualisation

Le mécanisme de *page cache* présente un conflit dans le monde de la virtualisation, où les ressources physiques sont fortement partagées et peu abondantes ; en ce qui concerne la *page*

cache, les suppositions sur la possession complète et exclusive de la mémoire par un système invité présentent un défi à une virtualisation efficace de la mémoire. Par ailleurs, toute la mémoire visible à une VM est en fait de la mémoire mappée par l'hyperviseur sur le système hôte (mémoire non libérable). Un artéfact de cette propriété est la redondance par double antémémoire (*double caching*), qui est une duplication de la cache du disque, soit dans la page cache du système invité (mémoire libérable sur le *guest* mais non libérable sur le *host*) et dans la page cache du système hôte (mémoire libérable). Bien que KSM (section 2.5.6) permette de remédier à la duplication de pages mémoires mappées, son travail ne s'étend pas à la *page cache* du système hôte puisque celle-ci n'est mappée à aucun espace mémoire. KSM n'est donc pas un remède à la redondance de la *page cache*. Singh (2010) a présenté les différentes méthodes et politiques pour l'utilisation de la page cache en systèmes virtualisés [39], particulièrement pour l'hyperviseur QEMU/KVM que nous rapportons ci-après.

1. **Guest only caching**
   Dans ce modèle, le maintien de la page cache est laissé au système invité et celle de l'hôte est contournée. Cette méthode peut aboutir à de la redondance de cache, les différentes VM accédant aux mêmes blocs sur disque garderont chacune une copie dans leur *page cache*. De plus, le système d'exploitation n'est plus responsable de contrôler les entrées-sorties vers le disque pour optimiser les accès à celui-ci et réordonner les requêtes. Finalement, tel qu'expliqué précédemment, la *page cache* de la VM est "libérable" du point de vue de celle-ci, mais mappée à l'hyperviseur comme étant non "libérable" pour l'hôte, ce qui rend le surengagement de la mémoire encore plus difficile à résoudre. D'un autre côté, cette méthode est requise pour permettre la migration de machines virtuelles avec QEMU. En effet, si la page cache est laissée à l'hôte, alors les pages mémoires qui la forment ne sont pas assignées au processus QEMU mais appartiennent au système d'exploitation hôte. Par conséquent, il est impossible de les migrer pour éviter la corruption de celle-ci, puisque sa page cache n'est pas mappée au processus.

2. **Host only caching**
   D'un autre côté il est aussi possible de configurer l'hyperviseur pour assigner la responsabilité de la *page cache* au système hôte. Les requêtes d'entrées-sortie des machines virtuelles passent donc par la mémoire du système hôte. Ceci présente souvent des avantages de performance, puisque celui-ci peut alors réordonner les requêtes vers les périphériques externes pour maximiser le débit de communication. Cela étant dit, il n'est pas possible de désactiver complètement la *page cache* du système invité. Il

est donc recommandé d'en réduire l'utilisation en assignant une valeur nulle à son paramètre `swappiness`. Nous mentionnons aussi que la cache de l'hôte peut être paramétrée davantage. QEMU supporte les modes *writeback* et *writethrough* pour la *page cache* de l'hôte pour des écritures vers le disque respectivement asynchrones et synchrones.

3. **Mixed caching**
   La configuration *mixed caching* permet à chacun des systèmes hôte et invité de maintenir une *page cache*. Les avantages et inconvénients mentionnés précédemment sont tout aussi applicables dans un tel modèle : duplication de la page cache, réordonnancement des requêtes d'entrées-sorties par le système hôte, etc.

2.6 Paravirtualisation des périphériques

2.6.1 Virtio

Tel que présenté précédemment, le noyau Linux s'intègre avec plusieurs hyperviseurs en tant que système invité. Virtio est un modèle permettant d'introduire une couche d'abstraction dans le but d'uniformiser les pilotes des périphériques d'entrée-sortie entre les différents hyperviseurs. De cette façon, chaque hyperviseur ne doit que supporter l'interface présentée par chacun des pilotes de la famille Virtio déjà intégrés au noyau invité pour en prendre avantage. Les développeurs d'hyperviseurs ne se voient plus obligés de réimplémenter des pilotes propres à leur VMM. L'approche de virtio est la séparation entre le pilote lui-même, la couche de transport (virtio utilise *virtio_ring* comme ring buffer) et la configuration dans le but d'atteindre une haute flexibilité [40]. Les pilotes de la famille Virtio sont chargés par le noyau invité (qui sont le *front-end*) et coopèrent avec l'hyperviseur qui, lui, implémente le back-end de chaque pilote. La communication entre le back-end et le front-end est donc elle aussi indépendante, et la couche de transport utilise le ring buffer de Virtio, vring, pour implémenter le modèle producteur–consommateur (invité–hôte). De cette manière, l'émulation d'un périphérique (le *back-end*) est spécifique à l'hyperviseur, mais pas son intégration au noyau invité (le *front-end*) dont la charge est prise par les pilotes virtio [41]. La figure 2.7 montre cette architecture. Celle-ci montre le pilote virtio-blk qui se charge de s'interfacer à un périphérique de type block tel qu'un disque. L'émulation du disque est prise en charge par Qemu, qui répond aux requêtes initiées par le pilote virtio-blk depuis la VM. La communication est implémentée par le ring buffer *virtio_ring* qui implémente le modèle producteur-consommateur. Le noyau Linux contient présentement plusieurs pilotes du modèle Virtio, tels que virtio-blk, virtio-net (pour l'émulation de la carte réseau), virtio-console

(pour un dispositif sériel), virtio-pci (pour l'émulation du contrôleur PCI).

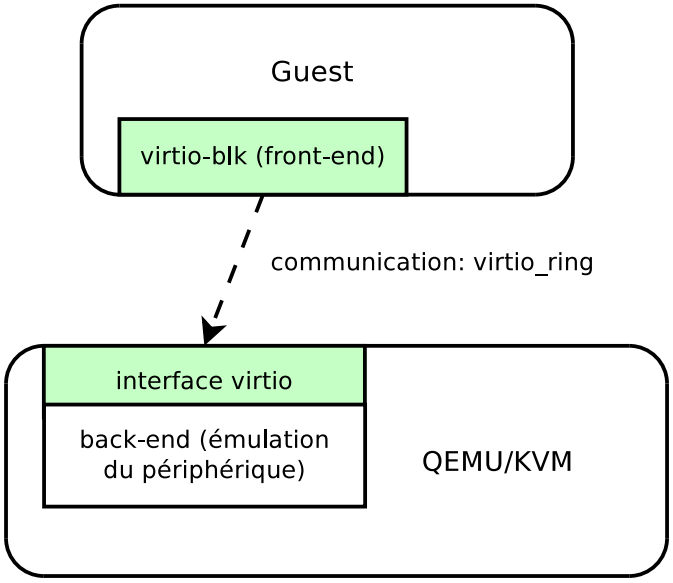

Figure 2.7 Architecture de communication du pilote de disque *virtio-blk*

Virtio-trace

Virtio-trace est un pilote récent de la famille virtio spécifique au traçage. Implémenté initialement pour Ftrace, il permet un transfert efficace de la trace du système invité au système hôte. Il fut introduit au noyau Linux en 2012 par Yunomae chez Hitachi, suivi d'un modèle de synchronisation pour machines virtuelles x86, tel que nous le présentons dans la section 2.7.2. Au lieu d'écrire la trace sur disque, celle-ci est transmise à l'hôte par l'intermédiaire de tubes. De plus, l'écriture dans les tubes de communication est effectuée via l'appel système `splice()`. Ce dernier permet un échange de données entre descripteurs de fichiers sans copie, pour un transfert efficace du contenu de la trace. En effet, la trace générée dans une machine virtuelle est généralement destinée à être analysée sur l'hôte par l'administrateur. Ce mécanisme évite de recourir au réseau pour transférer une trace noyau entre une machine virtuelle et son hôte. LTTng, à sa version courante 2.4, ne supporte pas encore l'intégration à ce pilote, bien qu'elle serait envisageable et avantageuse.

2.7 Notions de temps

Cette section se propose d'introduire les concepts liés à la gestion du temps. Tel que nous l'avons mentionné précédemment, la mise à jour de l'horloge tombe sous la responsabilité du

système d'exploitation. Nous présentons d'abord comment l'horloge est mise à jour et maintenue sous Linux ; nous introduisons ensuite une méthode existante pour la synchronisation de traces noyau de machines virtuelles.

2.7.1 Mesure et gestion du temps

Pour expliquer le mécanisme de mise à jour du temps sous Linux, nous commençons par présenter comment le traceur LTTng assigne les estampilles de temps aux évènements qu'il enregistre. Celui-ci utilise les fonctions `ktime_get()` pour le traçage en mode noyau, et `clock_gettime()` pour le traçage en espace utilisateur. Ces deux fonctions utilisent les mêmes variables en arrière-plan pour donner le temps du système, d'autant plus qu'une version VDSO [42] de `clock_gettime()` [43] a été implémentée pour x86. Leurs modes de fonctionnement sont similaires, et peut être décortiqué en analysant le code source du noyau. Linux utilise une structure de type `timespec` pour maintenir le temps. Celle-ci contient deux champs, l'un pour les secondes et l'autre pour les nanosecondes, et est mise à jour à chaque interruption du *system timer* en l'incrémentant par le temps écoulé depuis la dernière mise à jour (donc depuis la dernière interruption livrée par le *system timer*). Ce dernier représente une interruption matérielle livrée par un composant externe au CPU, dont le but est de fournir des interruptions différées. Une composante APIC (Advanced Programmable Interrupt Controller) se situe à proximité de chaque CPU et elle se charge de livrer des interruptions au CPU auquel elle est assignée. Celui-ci utilise un registre dans lequel il précise le délai avant la prochaine interruption, ce qui permet d'activer périodiquement des interruptions. La fréquence à laquelle ces interruptions sont livrées est définie dans Linux lors de la compilation du noyau par le paramètre de configuration `CONFIG_HZ`. Nous mentionnons que dans les systèmes plus anciens, un seul contrôleur avait cette tâche, le PIT (Programmable Interrupt Timer), dont les destinataires étaient l'ensemble des CPUs. Sur les systèmes d'exploitation *tickless* (voir `CONFIG_NO_HZ`), la fréquence de livraison des interruptions par le contrôleur APIC n'est pas régulière, et chaque interruption doit être spécifiée par le CPU. À des fins de réduction de consommation d'énergie, le CPU ne demande d'interruption au APIC que lorsqu'un processus lui est assigné par l'ordonnanceur. La livraison d'interruptions à un CPU inactif est généralement inutile. Ainsi, pour un CPU au repos, aucune interruption n'est livrée.

Lorsqu'une fonction de temps est appelée, `ktime_get()` et `clock_gettime()` retournent la valeur des secondes ajustée selon une interpolation sur les nanosecondes pour combler le temps écoulé depuis la dernière mise à jour.

2.7.2 Synchronisation des traces

Le plus grand obstacle à la corrélation des traces noyau des systèmes invités et hôte est la synchronisation de celles-ci. Notre analyse de préemption doit être effectuée sur l'union de plusieurs traces enregistrées en même temps sur des systèmes différents. Cependant, tel qu'expliqué dans la section précédente, le temps est mis à jour indépendamment dans les différents systèmes. Une certaine marge d'erreur existe toujours, et la combinaison des évènements de diverses traces en ordre chronologique ne résulte pas toujours en un flot d'exécution cohérent. Par exemple, un évènement dans une VM peut avoir une estampille de temps légèrement supérieure à celle d'un évènement d'ordonnancement qui "enlève" cette VM du CPU, ce qui résulte en une analyse erronée. Tel que présenté par [44], le TSC peut être utilisé pour remédier à ce problème. Le TSC, ou TimeStamp Counter, est un registre spécifique à la famille x86 qui compte le nombre de cycles CPU depuis le démarrage de la machine, et peut être utilisé à des fins de mesure de temps à haute précision. Lorsqu'il est lu depuis un système virtualisé par assistance matérielle, la valeur du TSC est automatiquement décalée en matériel par le CPU par la valeur du champ `TSC_OFFSET` de la VMCS. En utilisant le traceur ftrace, les estampilles de temps attribuées à chacun des évènements enregistrés utilisent le TSC. En suivant les modifications du champ `TSC_OFFSET`, il est possible de ramener les évènements de la trace du système invité à la même ligne de temps que celle de l'hôte, en assurant une cohérence de la trace résultante. Cependant, cette approche présente des problèmes en terme de portabilité et de facilité d'utilisation. Tout d'abord, le registre TSC est spécifique aux architectures x86, ce qui rend son utilisation impossible sur d'autres architectures. Ensuite, l'enregistrement de la trace doit commencer avant le démarrage de la machine virtuelle pour obtenir l'évènement initial de l'assignation du `TSC_OFFSET`. De plus, la perte potentielle d'évènements lors du traçage rend cette méthode inefficace, à cause du risque de perdre un évènement de modification au `TSC_OFFSET`, il est donc aussi impossible d'interrompre le traçage ce qui pourrait poser un problème sur des environnements de production.

CHAPITRE 3

MÉTHODOLOGIE

Cette section présente les aspects méthodologiques de ce projet de recherche. Tel que nous l'avons démontré jusqu'ici, la virtualisation impose plusieurs contraintes qui font de la surveillance des machines virtuelles un travail ardu. Après avoir établi le contexte du travail, nous commencerons par présenter les détails de la méthode de synchronisation de traces proposée. Cette partie se veut complémentaire à l'explication de synchronisation dans l'article de la section 4, qui, elle, est plus brève. Par la suite, nous montrons quelques notions pratiques pour reproduire les expériences, comme les points de trace à activer et la configuration de QEMU. Finalement, nous présentons les contributions complémentaires au contenu de ce mémoire.

3.1 Contexte de travail

Dans QEMU/KVM, chaque machine virtuelle est un processus QEMU, et chacun de ses CPUs virtuels est émulé par un fil d'exécution (*thread*) appartenant au processus. Cette approche simplifie grandement l'analyse des machines virtuelles actives. En effet, il est possible de suivre l'état d'un vCPU en suivant simplement l'état du fil d'exécution qui l'émule depuis la trace du système hôte. Similairement, il est possible de suivre l'état des processus qui s'exécutent sur ce vCPU depuis la trace du système invité. L'union des deux traces permet alors d'établir les interactions entre les processus du système hôte et ceux de la VM. La figure 3.1 montre l'architecture de traçage requise pour suivre une telle méthode de travail. Les systèmes des machines virtuelles peuvent être analysés si le traceur est installé dans chacune de ceux-ci, alors que l'information sur KVM et le noyau hôte est fournie par le traceur du système hôte. Une fois les traces générées, elles sont unies et analysées à postériori, en mode *offline*.

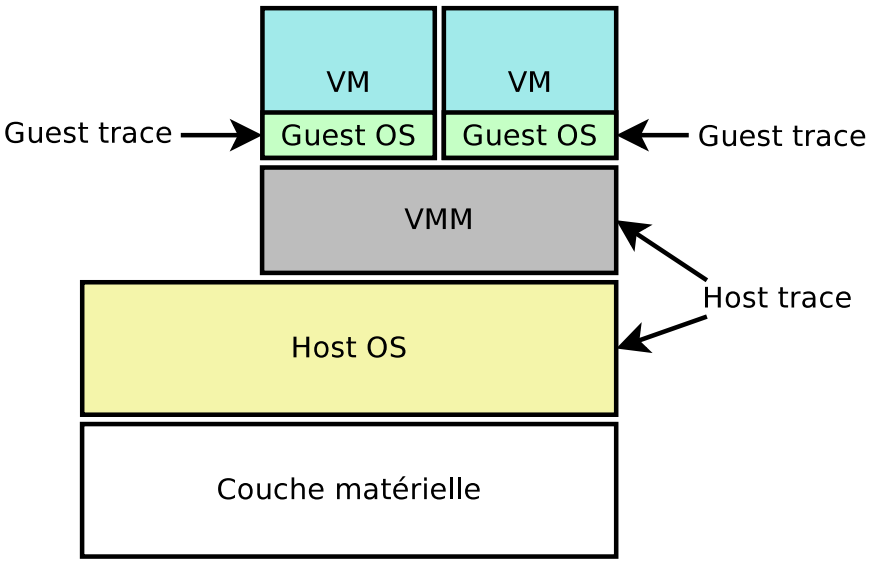

Figure 3.1 Architecture de traçage requise pour reconstruire l'état global du système

3.2 Besoin de synchronisation

Nous avons déjà mentionné que chaque système d'exploitation est responsable de sa propre gestion du temps. Nous montrons dans la section présente les résultats d'expériences complémentaires qui permettent de concrétiser le décalage et l'incohérence du maintien du temps. Les courbes sont obtenues en utilisant la formule

$$y = t_H - t_G \tag{3.1}$$

où t_H est une estampille de temps du système hôte et t_G est l'estampille de temps qui lui est équivalente du système invité. La figure 3.2 montre la différence entre les horloges du système hôte et d'une machine virtuelle inactive. La courbe obtenue montre deux indications qui doivent être prises en compte lors de l'analyse. Tout d'abord, nous constatons une tendance de déviation entre les horloges. En d'autres termes, les horloges des deux systèmes s'éloignent l'une de l'autre au fur et à mesure de l'exécution (*time drift*). Ensuite, la fluctuation de la courbe montre que la différence entre les temps est fortement variable. Ce deuxième artéfact est dû à la nature des noyaux *tickless* qui désactivent les interruptions du *system timer* sur les CPU inactifs. Comme la machine virtuelle est globalement inactive, la livraison des interruptions est non régulière. Les répercussions de cette caractéristique sont, entre autres, une irrégularité de la fréquence de mise à jour du temps. Pour vérifier cette hypothèse, nous effectuons la même expérience sur une machine virtuelle dont le CPU est actif à 100% ; le résultat est montré dans la figure 3.3. Nous constatons la même tendance de déviation des

horloges, mais la variabilité entre celles-ci est moins importante. Ceci est explicable par le fait que les horloges sont mises à jour régulièrement sur les systèmes hôte et invité puisque les interruptions du *system timer* leur sont livrées périodiquement. Nous notons aussi la présence de deux échantillons éloignés de la courbe de tendance. Ces valeurs sont aberrantes et peuvent statistiquement être ignorées. Cependant, ce comportement peut être dû à une imprécision du *timer* de l'hôte qui émule le module APIC de la machine virtuelle.

Nous remarquons aussi que la différence dans la figure 3.3 est négative, ce qui indique que le temps dans la VM semble passer plus rapidement, contrairement aux résultats de la première expérience. Finalement, la pente de la courbe est due au fait que le temps semble être mis à jour plus rapidement dans un système que dans l'autre.

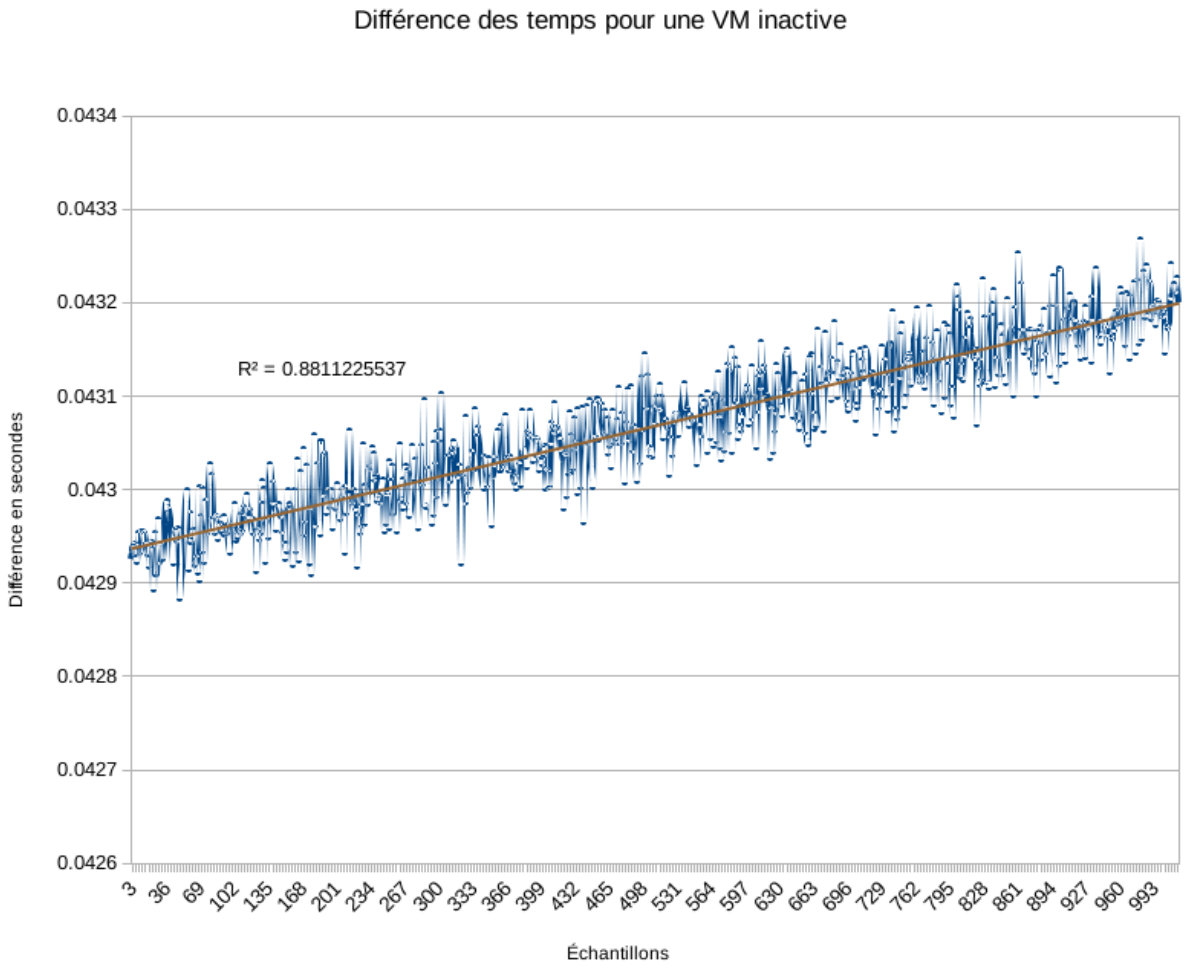

Figure 3.2 Différence entre les horloges des systèmes invité et hôte en fonction du temps pour une VM inactive

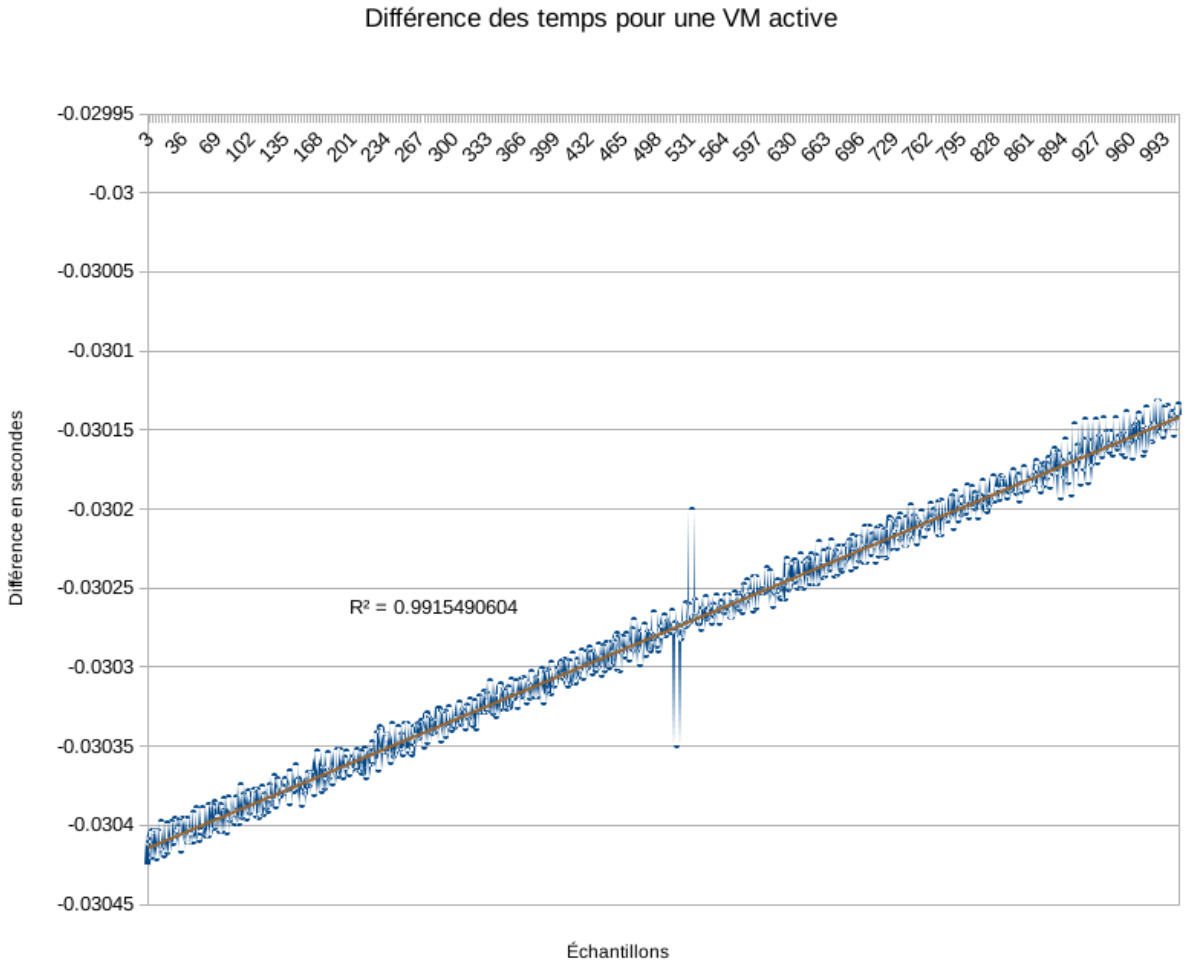

Figure 3.3 Différence entre les horloges des systèmes invité et hôte en fonction du temps pour une VM active

3.3 Approche utilisée pour la synchronisation de traces

3.3.1 Algorithme de base

Jabbarifar (2014) a proposé une méthode pour la synchronisation de traces de systèmes distribués [45]. Cet algorithme, nommé "*Fully incremental convex hull synchronization algorithm*", est basé sur l'appariement d'évènements. En d'autres termes, un évènement a d'une trace T_1 doit être jumelé à un évènement b d'une autre trace T_2 par la relation a génère b, ou a provoque b, que nous notons comme suit :

$$a_{T_1} \rightarrow b_{T_2} \tag{3.2}$$

Ainsi, une borne inférieure de temps est imposée sur l'évènement b puisqu'il est impossible que celui-ci se produise avant a. Dans le but d'imposer une borne supérieure aux évènements de T_2, la même correspondance entre une paire d'évènements de chaque trace doit être effectuée, mais dans le sens inverse ; c'est-à-dire qu'un évènement c de T_2 doit générer un évènement d de T_1 ($c_{T_2} \rightarrow d_{T_1}$).

Une fois les noms des évènements à apparier fournis à l'algorithme, celui-ci produit une formule de synchronisation qui est fonction de l'écart entre les deux horloges et de la dérive de celles-ci. Cette formule est ensuite appliquée à toutes les estampilles de temps de T_2, la ramenant à la même échelle de temps que T_1. Pour pouvoir être appariés, a et b doivent contenir dans leur charge utile une même clé k unique qui permet d'effectuer une correspondance de type 1-à-1 ($a_{T_1} \overset{k}{\rightarrow} b_{T_2}$).

Dans l'implémentation initiale de l'algorithme, le traçage de transmission de paquets TCP entre des systèmes était requis. En effet, un numéro de séquence associé à chaque paquet est présent dans la charge utile des évènements. L'émission d'un paquet sur un système (a_{T_1}) provoque sa réception sur l'autre (b_{T_2}). Nous précisons aussi que la synchronisation des traces s'effectue par paires. En d'autres termes, chacune des traces est synchronisée selon une trace dite hôte dont l'horloge est la référence. En conséquence, le nombre de traces à synchroniser n'affecte en rien la précision de l'algorithme.

3.3.2 Implémentation des évènements à apparier

L'article du chapitre 4 fournit de plus amples détails sur le fonctionnement de cet algorithme. Nous présentons ici les détails de l'implémentation des paires d'évènements servant à la synchronisation. En effet, aucune paire de points de traces ne répond aux critères établis par l'algorithme de synchronisation. Certains évènements respectent la relation $a \rightarrow b$. Par exemple, lorsque le système invité programme une interruption sur le module APIC, cette instruction est trappée par l'hyperviseur qui l'émule à l'aide d'une minuterie à haute résolution (*hrtimer*). L'écriture dans le registre de programmation d'une interruption par le *guest* ainsi que la création d'une minuterie par le *host* sont tous deux des évènements instrumentés. Cependant, ils ne partagent aucune clé commune qui permette de les apparier avec une relation 1-à-1. Il en est de même pour le couple dans le sens inverse. Lorsque la minuterie expire sur le *host*, celui-ci émule l'interruption matérielle du module APIC de la VM en y injectant une interruption virtuelle (*kvm_inj_virq*). L'injection de l'interruption ainsi que la reconnaissance de cette interruption par le *guest* sont aussi instrumentées. Cependant, aucune clé unique commune n'est partagée par ces points de trace.

Pour répondre au requis de clé unique partagée entre les évènements, nous avons ajouté deux modules à LTTng sous forme d'*addons* dans une branche de développement expérimentale. Chacun des deux modules est destiné à être chargé par un système d'exploitation, soit hôte, soit invité. Lorsqu'un noyau invité charge le module *lttng-vmsync-guest*, celui-ci enregistre une *probe* au point de trace *softirq_exit*. En d'autres termes, à chaque invocation du point de trace *softirq_exit*, une fonction du module *lttng-vmsync-guest* sera exécutée. La raison pour laquelle nous invoquons la fonction de synchronisation uniquement lors de la sortie d'un SoftIRQ est pour limiter son taux d'invocation, tel qu'expliqué dans l'article à la section 4. La fonctionnalité de celle-ci est simplement de générer un point de trace (a) et d'effectuer une invocation explicite à l'hyperviseur par le biais d'un *hypercall*. Similairement à un appel système, il est possible de passer des arguments à l'hyperviseur en les affectant aux registres `rax`, `rbx`, `rcx` et `rdx`. Dans notre cas, l'argument passé en paramètre via le *hypercall* est la valeur d'un compteur interne au module. Le point de trace a généré à priori contient également ce compteur dans sa charge utile.
Similairement, le module *lttng-vmsync-host* enregistre une probe au point de trace *kvm_hypercall*. Le rôle de cette *probe* est de générer un point de trace (b) avec la valeur de l'argument reçu à travers le *hypercall* dans sa charge utile. Nous avons à présent répondu aux critères requis par l'algorithme de synchronisation ; $a_G \xrightarrow{k} b_H$ où a_G est le point de trace généré par le *guest* juste avant le *hypercall*, b_H est l'évènement de reconnaissance du *hypercall* sur le *host* (*acknowledgement*) et k est la valeur du numéro de séquence (le compteur) passé via le *hypercall* et inclus dans les deux points de trace.

Avant de redonner la main à la VM, le module *lttng-vmsync-host* génère un évènement (c) qui indique la fin de son travail. Dès que la machine virtuelle poursuit son exécution, après l'instruction `vmcall`, elle génère un évènement qui indique la reprise de l'exécution (d). Malheureusement, aucun mécanisme de partage de données n'est possible de l'hyperviseur vers la VM. Cependant, le passage de paramètre n'est pas requis dans ce cas-ci. En effet, c et d inscrivent tous deux la valeur k + *1* dans leur charge utile. Cette approche est valide puisque la séquence d'évènements est forcément a, b, c, d. Les deux paires d'évènements, en sens opposés, qui permettent la synchronisation sont donc les suivantes :

$$a_G\{k\} \xrightarrow{k} b_H\{k\} \tag{3.3}$$

$$c_H\{k+1\} \rightarrow d_G\{k+1\} \tag{3.4}$$

La figure 3.4 montre la séquence des points de trace.

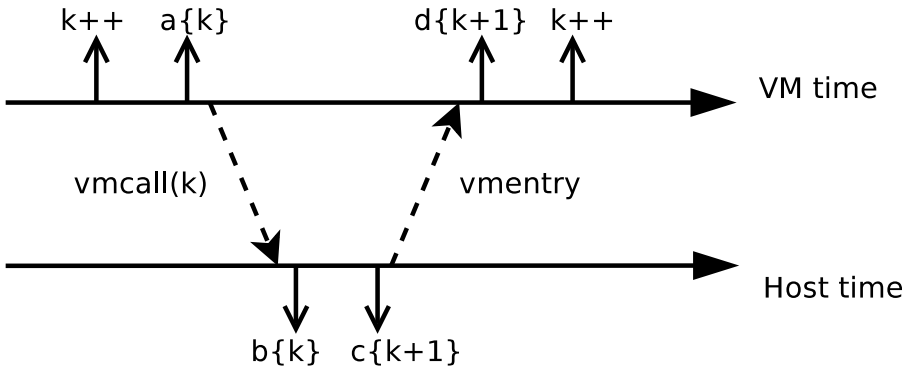

Figure 3.4 Séquence des évènements de synchronisation entre les systèmes hôte et invité

Nous mentionnons que l'algorithme pourrait être légèrement modifié pour exiger moins d'évènements, en gardant la même précision de synchronisation. Par exemple, b et c pourraient être unis en un unique évènement z, et la relation entre les évènements serait :

$$a_G\{k\} \xrightarrow{k} z_H\{k\} \rightarrow d_G\{k\} \tag{3.5}$$

Ainsi, des bornes supérieure et inférieure sont imposées sur les évènements de la trace du *guest*.

3.4 Modèle de traitement des traces

Tel qu'expliqué précédemment, l'analyse des traces se fait à travers un arbre d'attributs qui décrit le système. Au fur et à mesure du traitement d'une trace, les attributs sont modifiés selon des règles précises. La formulation de ces règles représente l'analyse en soi. Le chapitre 4 présente en détail ces règles ainsi que la façon avec laquelle elles ont été établies. Les traces issues des différents systèmes doivent être générées avec certains points de trace spécifiques activés. Les évènements les plus importants pour obtenir une analyse minimale cohérente sont *sched_switch* (pour tous les systèmes tracés), *kvm_entry* et *kvm_exit* (système hôte uniquement). Pour une analyse plus riche et détaillée, les évènements *sched_process_fork*, *sched_process_free* et *sched_migrate_task* peuvent également être activés. Ces points de trace signifient respectivement la création d'un nouveau processus, sa terminaison, et sa migration d'un CPU à l'autre. Le point de trace *sched_process_wakeup* indique qu'un processus sort de l'état bloqué et a été inséré dans la liste des processus prêts à s'exécuter. Lorsqu'activé, le délai entre cet évènement et le début d'exécution du processus présente une métrique intéressante, connue sous le nom de *wake-to-schedule-in time* [8], qui indique le temps d'attente

du processus entre la demande du CPU et son acquisition. Les quatre évènements de synchronisation que nous avons ajoutés doivent être activés également, cependant ils n'affectent en rien l'analyse dans le sens où ils ne décrivent aucun changement d'état des processus.

Il est aussi intéressant de mentionner une configuration particulière de QEMU que nous avons choisie pour ce projet. Tel qu'énoncé précédemment, chaque machine virtuelle QEMU/KVM est un unique processus du point de vue de l'hôte. Par conséquent, pour n machines virtuelles actives, n processus sont créés. Cependant, comme pour la plupart des processus sous Linux, leur nom est identique à celui du fichier binaire qu'ils exécutent, à savoir, `qemu-system-x86_64`. Il est à priori impossible de les dissocier les uns des autres et de savoir quel processus émule quelle VM. Ceci pose un problème majeur lors de l'analyse de traces. Il est possible de configurer qemu pour inclure le nom de la machine virtuelle dans le nom du processus. En activant l'option `set_process_name = 1` dans le fichier `/etc/libvirt/qemu.conf`, les noms des processus QEMU prennent la forme `qemu:nom-vm` où *nom-vm* est le nom de la machine virtuelle que le processus sert.

3.5 Contributions pertinentes

Diverses contributions ont été accomplies dans le cadre de ce projet. Tout d'abord, ce travail a mené à la rédaction d'un premier article de conférence sur lequel est basé l'article du chapitre 4, intitulé "Virtual Machines CPU Monitoring with Kernel Tracing". Cet article fut présenté à la conférence CCECE (Canadian Conference on Electrical and Computer Engineering) de l'IEEE.

Ensuite, nous avons proposé des modifications à l'hyperviseur QEMU maintenu par Red Hat pour supporter le traçage en mode utilisateur avec LTTng. En effet, QEMU est instrumenté par des points de trace et peut être analysé en utilisant différents traceurs utilisateurs tels que DTrace. Il peut même être configuré pour un mode de traçage sur la sortie d'erreurs (*stderr*). Les contributions pour supporter LTTng UST 2.x ont été acceptées et sont déployées dans les versions récentes de QEMU. Cependant, ce dernier n'est pas configuré par les mainteneurs de distribution pour utiliser LTTng comme traceur par défaut. Une configuration et compilation manuelles sont donc requises.

Finalement, des modifications à LTTng, qui est maintenu par Efficios, ont été proposées pour supporter les points de trace de KVM spécifiques à l'architecture x86. Ces modifications ont aussi été acceptées et intégrées à LTTng.

CHAPITRE 4

ARTICLE 1 : FINE-GRAINED PREEMPTION ANALYSIS ACROSS VIRTUAL MACHINES

Authors

Mohamad Gebai
École Polytechnique de Montréal
mohamad.gebai@polymtl.ca

Francis Giraldeau
École Polytechnique de Montréal
francis.giraldeau@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

4.1 Abstract

This paper studies the preemption between programs running in different virtual machines on the same computer. One of the current monitoring methods consists into updating the average steal time through collaboration with the hypervisor. However, the average is insufficient to diagnose abnormal latencies in time-sensitive applications. Moreover, the added latency is not directly visible from the virtual machine point of view. The main challenge is to recover the cause of preemption of a task running in a virtual machine, whether it is a task on the host computer or in another virtual machine.

We propose a new method to study thread preemption crossing virtual machines boundaries using kernel tracing. The host computer and each monitored virtual machine are traced simultaneously. We developed an efficient and portable trace synchronization method, which is required to account for time offset and drift that occur within each virtual machine. We then devised an algorithm to recover the root cause of preemption between threads at every level. The algorithm successfully detected interactions between multiple competing threads in distinct virtual machines on a multi-core machine.

4.2 Introduction

Cloud environments present advantages of increased flexibility and reduced maintenance cost through resource sharing and server consolidation [46]. However, virtual machines (VMs, or guests) on the same host computer may compete for shared resources, introducing undesirable latency. Previous study found that jitter is affecting response time of programs on popular commercial cloud environment [47]. In cloud environments, virtual machines have the illusion of absolute and exclusive control over the physical resources. However, the host's resources are more often than not overcommitted, whereas they appear to guest operating systems as being more available than they actually are [36]. As a result, virtual machines on the same host computer may interfere with each other without their knowledge, inducing invisible yet real latency.

The diagnosis is more complex when the guest is isolated from its external environment and an additional virtualization layer is introduced. It is therefore necessary to have powerful and efficient tools to diagnose the root cause of unexpected delays when they occur in a virtualized environment. To our knowledge, no such tool was available.

This study focuses on processor multiplexing across virtual machines. In particular, we are interested in automatically identifying the root cause of task preemption crossing virtual machines boundaries. The challenge is to consider the system as a whole, while preserving virtual machine isolation. Flexibility and portability constraints are also important for practical considerations. The approach should be independent from the architecture to account for portability, whereas flexibility requires independence from the hypervisor and the tracer.

The approach we propose is based on kernel tracing, which is an effective and efficient way to debug latency problems [48]. The method we propose consists into aggregating kernel traces recorded simultaneously on the host and each virtual machine. However, as each operating system is responsible of its own timekeeping, timestamps from different traces are not issued using the same clock source. As a result, trace merging produces incoherent results without a method to synchronize traces.

Three main contributions are presented in this paper. First, we propose an approach for trace synchronization. At the trace merging step, we propose an algorithm that modifies timestamps of the guests' traces to bring them back to the same timespan as the host. Secondly, we implemented an analysis program that transforms aggregated kernel traces to a graphical view that shows the states of the virtual machines and their respective virtual CPUs (vCPUs) through time while taking into consideration virtualization and its impact. Thirdly, we implemented an additional analysis program that presents the interactions of threads across different systems. Such an analysis can be performed by recovering the execution flow

centered around a particular thread.

The rest of this paper is structured as follows : Section 4.3 goes through different approaches currently used for virtual machine monitoring. Section 4.4 introduces the required concepts in virtualization and tracing, and states the problem addressed by this paper. Section 4.5 explains our approach for trace synchronization at the aggregation step. Each of sections 4.6 and 4.7 introduces an analysis module and its inner working. Section 4.8 shows some representative use cases and their analysis results. Section 4.9 concludes.

4.3 Related Work

On Linux kernels supporting paravirtualization, `top` reports a metric specific to virtual machines, named *steal time*. This metric shows the percentage of time for which a vCPU of the VM is preempted on the host. While this information can give a general idea or a hint of preemption, it does not report the actual impact on the running threads nor the source of preemption. Moreover, `top` adds significant overhead as it gathers information by reading entries in the `proc` pseudo-filesystem, and offline analysis or replay of the execution flow are not possible.

Perf has been extended to support KVM tracing. Using its subcommand "`kvm`", one can use perf to get statistics about latency introduced by the hypervisor involvement throughout the execution of a VM. However, detailed CPU usage, preemption and virtual machines interactions are not part of perf's extent.

Shao et al. use an approach based on tracing within Xen to generate useful metrics for virtual machines [8]. Based on scheduling events, latency due to virtual CPU preemption can be easily calculated. Other metrics of interest are also presented such as the wake-to-schedule time. However, these metrics are mostly useful for analyzing Xen's scheduler, which doesn't apply to KVM as it is a an "extension" to the kernel and thus uses its scheduler. Moreover, the impact of virtual machines on one another can not be retrieved from Xen traces.

In [49] and [50], the authors use profilers like `perf` to monitor resource usage of virtual machines. Metrics like CPU usage and hypervisor involvement can be retrieved from profiling. Again, the investigations of causes of preemption and specific thread analysis are not possible.

In [51], an approach for PMU virtualization using `perf` is proposed. The authors use hardware performance counters to monitor virtual machines. While these metrics can be useful in a lot of cases, information about vCPU preemption can not be retrieved or analyzed using profiling. However, we think it could be a complementary work to show additional in-depth execution information.

As for trace synchronization, previous studies [8, 44] have used the TSC as a common

time reference to approach timekeeping and clock drifts issues. The TSC is a CPU register on x86 architectures which counts CPU cycles since the boot (uptime) of the system. When read from a virtualized system, the TSC is automatically offset in hardware to reflect the uptime of the guest operating system. The value of the offset is specified by the `TSC_OFFSET` field in the VMCS. Each VM on the host has its own `TSC_OFFSET` value, and reading the TSC from different systems always returns a coherent value with respect to the boot time of said system. Once traces are recorded on different systems, converting guest TSC values to host TSC values comes down to subtracting the value of `TSC_OFFSET` from each timestamp. However, the TSC offset may have to be adjusted during the execution of the VM upon certain events, such as virtual machine migration. As a result, `TSC_OFFSET` adjustments have to be tracked down by the tracer at run-time. If tracing is not enabled before the creation of the virtual machine, the initial value of the TSC offset cannot be obtained, unless explicitly requested by the tracer. Additionally, this approach does not allow for the possibility of lost events since a TSC adjustment event could be lost. In any manner, synchronization by TSC offsetting does not meet our requirement of portability, as it is an x86-specific register. Moreover, TSC offsetting is specific to hardware-assisted virtualization, thus it cannot be used with other virtualization methods, which does not meet our flexibility requirement. Additionally, the TSC register only counts CPU cycles since boot time, which is not as meaningful as an absolute wall clock time, especially on computers with a non-constant TSC where the conversion from TSC to real time would be an additional challenge (CPU flag `constant_tsc` can be queried to verify this property).

4.4 Definitions and Problem Statement

4.4.1 Hypervisor

The hypervisor used in this study is QEMU/KVM. KVM [23] is included in Linux as a kernel module. QEMU is the userspace component that interacts with KVM to take advantage of hardware assistance introduced by microprocessor vendors. Moreover, as KVM is part of the Linux kernel, it can take advantage of its tracing infrastructure. KVM is instrumented with tracepoints which can be traced using any kernel tracer. In QEMU/KVM, each virtual machine is a QEMU process, and each of its virtual CPUs (vCPUs) is emulated by a separate thread that belongs to that same process.

On hardware-assisted virtualization, the CPU transits between non-root and root modes. The former is entered using the `vmentry` instruction by the hypervisor, giving control to the VM's native code. The later is reached when the VM executes an instruction that triggers a trap (such as a sensitive instruction or writing to a privileged register), which allows the

hypervisor to take control of the execution and react to the trapped instruction (usually with emulation). Moreover, a data structure called VMCS (Virtual Machine Control Structure) [30] contains the state of a virtual machine. This state is used as an interaction mechanism between the VM and the hypervisor [52], as well as to define define behavioral elements, such as enabling or disabling hardware TSC offsetting.

In this article, the terms VMM and hypervisor will be used interchangeably. The same applies for terms VM, guest system and virtualized system.

4.4.2 Trace indexing

We implemented our trace analysis algorithm using the Tracing and Monitoring Framework (TMF) which is deployed as an Eclipse plugin [6]. TMF indexes the trace using a State History Tree (SHT) [7]. The SHT represents the state of the whole system, and is updated at each event to define time intervals [53]. This index allows efficient stabbing queries, returning the complete state of the system at a given time. A node of the tree is a key-value pair, where the key is a path component, and the value is an attribute associated with a duration, that gets updated as the trace is being processed. The rules, by which attributes are updated, are established by our algorithm presented in section 4.6.

Our algorithm requires kernel traces from all systems in the setup, i.e., the host and guests operating systems. Events from these traces are then merged and sorted by chronological order for processing. TMF reads the trace one event at a time and modifies the SHT attributes. Figure 4.1 shows a part of our SHT. For instance, the path "`/Virtual Machines/Ubuntu/CPUs/CPU0/Current Thread`" contains the
thread ID executing on CPU 0 of the VM named "Ubuntu". When a scheduling event such as *sched_switch* from the VM's trace is processed, the value of the attribute is changed from the TID of the former thread to the latter's. Similarly, the attribute at path "`/Host/Threads/Thread 1234/Status`" holds the status of the thread whose TID is 1234 on the host. This attribute may be modified when a context switch event involving the thread 1234 is being processed.

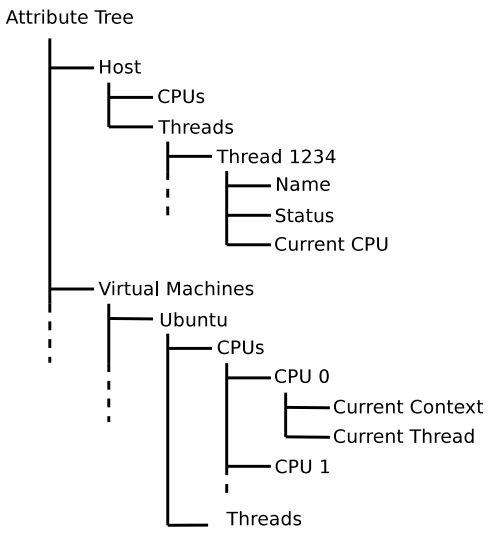

Figure 4.1 Example of a state history tree

4.4.3 Relevant Tracepoints

In this study, we use LTTng as a kernel tracer. LTTng was designed for high throughput tracing while reducing as much as possible its impact on the traced system [3]. We now introduce the key tracepoints for our analysis. We present their signification, as well as the content of their respective payload. This section is complementary to section 4.6 which explains how these tracepoints are used to update the SHT.

The `sched_switch` tracepoint indicates a context switch on the CPU which recorded the event. Useful payload fields are the names and the TIDs (Thread Identifiers) of the former and new threads involved in the context switch. Since all events are timestamped using the system's time at the nanosecond scale, the amount of time spent on each CPU by a specific thread is easily computed by subtracting the timestamps of the `sched_switch` events involving a particular thread.

Tracepoint `sched_migrate_task` indicates the migration of a thread from one CPU to another. Its payload holds the TID of the migrated task, as well as the origin and the destination CPU identifiers. Tracepoint `sched_process_fork` indicates the creation of a new process, and exposes the names, PIDs (Process Identifiers) and TIDs of the newly created process as well as its parent's. Its complementary event, `sched_process_exit`, records the end of life of a thread. The payload contains the name and TID of the process.

VMX mode transitions by KVM can be tracked by enabling the `kvm_entry` and `kvm_exit` events. Tracepoint `kvm_entry` indicates a transition from root to non-root modes, and thus the beginning of the execution of the VM's native code. On the other hand, tracepoint

`kvm_exit` indicates the opposite transition, which interrupts the execution of the VM and gives control to KVM. Elapsed time between consecutive `kvm_exit` and `kvm_entry` events represents overhead introduced by the hypervisor.

4.4.4 Addressed Problem

As mentioned in section 4.2, investigating latency problems in virtualized systems is a non-trivial task. The isolation of virtual machines from their environments imposes limits on the scope of traditional analysis tools. Moreover, the virtualization layer itself adds overhead due to the involvement of the hypervisor for privileged operations [33]. Furthermore, the assumption of exclusive access to the hardware layer by each virtual machine inevitably induces hidden latency due to the overcommitment of resources, particularly the CPU. In this chapter, we explain how we used kernel traces recorded in each VM and on the host simultaneously to investigate such problems. As we present in section 4.8, the tools resulting from our study help the users to easily find the latency cause due to CPU sharing among virtual machines, as well as the actual threads that affect the completion time of a certain workload. However, the merging of distributed traces is a problem in itself as each operating system is solely responsible of its own timekeeping. The next section presents our first contribution which is to achieve trace synchronization.

4.5 Trace Synchronization

4.5.1 Event matching

LTTng uses the monotonic clock of the kernel for timestamping events, rather than the raw Time Stamp Counter (TSC). It avoids architecture-dependent limitations inherent to the TSC, such as TSC synchronization between cores and non-constant TSC on variable frequency CPUs. Even in the case of an ideal TSC (invariant and synchronized between cores), the value is dependent on the processor frequency, and thus needs to be scaled for the user. Therefore, the monotonic clock internally scales the TSC to nanoseconds and applies an offset to represent the current time.

In addition, the monotonic clock guarantees total ordering, even in the case of modification of the system's wall clock time while tracing, and therefore is an ideal source for event timestamps. Moreover, the TSC is an x86-specific register and using it as a clock source does not meet our requirement of portability.

The monotonic clock is given using the simplified equation

$$t = T + tsc \times x \tag{4.1}$$

where T is a coarse-grained value updated upon system timer interrupts, adjusted using the TSC to account for the elapsed time since the last value update (last timer interrupt). Although the TSC is paced at the same rate across the different virtual systems, the offset values T of each system are not, and thus are subject to drifting apart as time goes by. In fact, modern tickless operating systems disable timer interrupts on idle processors to reduce energy consumption. As a result, the update period of T is variable, which may contribute to increase the time difference between systems. Furthermore, virtual machines may be set to different timezones, introducing even more incoherent timestamping when traces are merged together, which would make them appear as being recorded at different moments. Moreover, high precision timestamping and clock drifting do not allow for simple clock offsetting to ensure coherency between traces. This section presents our approach to ensure coherent trace merging.

We use the fully incremental convex hull synchronization algorithm to achieve offline trace synchronization, introduced by [45] for distributed traces synchronization. Each guest trace is processed individually and synchronized according to the host's trace whose timeline is taken as a reference. This approach is based on event matching between two traces. In order to use the synchronization algorithm, an event a from one trace must be associated to another complement event b from the other. Each couple of events $\{a, b\}$ must respect the following equation :

$$a_{T_1} \xrightarrow{k} b_{T_2} \tag{4.2}$$

In other words, the following requirements have to be met :

1. Causality : a must (quickly) trigger b ;
2. Bijection : a and b must share a common and unique key k in their payloads ;
3. Every event b must be either unmatched or matched to a single event a. Unmatched events b are ignored.

The key k is used to match a and b with a one-to-one relation. A lower delay between events a and b results in a more precise synchronization scheme. A synchronization formula is then derived by the algorithm which is a function of clocks offset and time drift. This formula is then applied to all timestamps of the guest's trace, bringing them to the same timebase as the host. By using the relation "a triggers b", a lower bound is imposed on the timestamps of events b as they cannot appear before their matching event a. Events between two consecutive events b are then adjusted to respect this constraint. With that being said, an upper bound has to be imposed as well to events b as clocks are subject to drifts. To set an upper bound on events b, we use the same matching approach in the opposite direction between systems. If a is an event in the guest OS that triggers b on the host, then an event c

on the host that triggers an event d on the guest is needed. Figure 4.2 shows events a, d, b, c from the original trace correctly reordered as a, b, c, d after the synchronization.

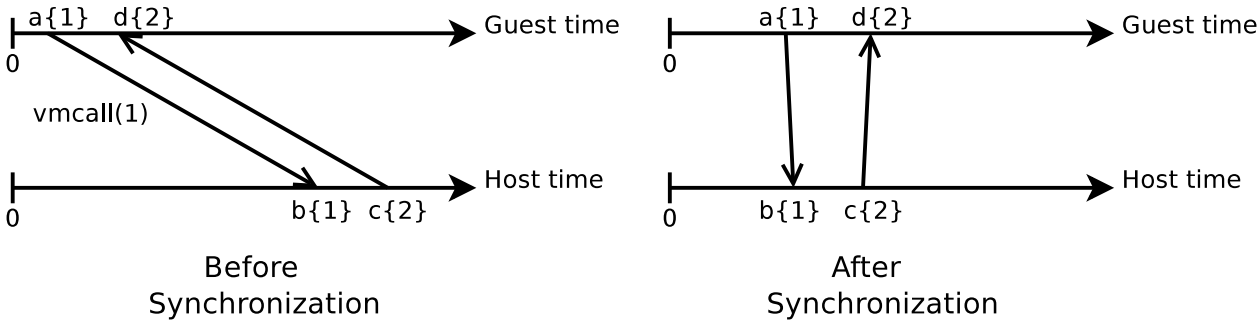

Figure 4.2 Lower and upper time bounds with matching events are used to synchronize traces

4.5.2 Implementation in virtualized systems

Previous section 4.5.1 explained the general approach of the fully incremental convex hull algorithm for trace synchronization. However, the requirements for this algorithm are not directly met in our case. Originally, the algorithm was built based on TCP packet exchange events : `send` and `receive` events respectively as a and b (`send` triggers a `receive` and the TCP packet number is the key k). In Cloud environments, virtual machines do not necessarily exchange TCP packets with each other or with the host, as VMs are usually provided to different clients. As a result, we need to customize the setup of the virtual machines to generate events both in the VMs and on the host that would respect the requirements established earlier. This section introduces our approach to obtain events that can be used to achieve trace synchronization.

We added tracepoints to the kernel through a loadable module for flexibility, so no modification to kernel code would be required to perform trace synchronization. Upon loading, this module registers a probe to the system timer's interrupt. In other words, every time the system timer issues an interrupt to the CPU, our synchronization routine will be invoked. The synchronization routine can be summed up as follows :

— Guest : Trigger hypercall (event a)
— Host : Acknowledge hypercall (event b)
— Host : Give control back to the VM (event c)
— Guest : Acknowledge control (event d)

The first pair of events (a, b) can be simulated by issuing a hypercall. When executing the `vmcall` instruction from the guest OS (event a), a trap is generated by the CPU and

control is given to the hypervisor, which in turn acknowledges the trap (event b). A counter X is passed to the host OS as a parameter to the hypercall. This parameter will serve as the shared key required by the synchronization algorithm. As a result, events a and b are both recorded in a short period of time on the guest and host OS respectively, both holding the same value X as their payload.

Simulating the pair of events (c, d) is not as trivial since different constraints are imposed on the host-to-guest communication. No mechanism of parameter transmission is easily accessible in such a communication. Implementing shared memory between the guest and host is too intrusive as it would add too much complexity to both systems, and would probably require modification to both kernels. However, execution of the guest's code continues naturally after involvement from the hypervisor. We can take advantage of this property to simulate a parameter transmission when the execution returns from hypercall handling. Event c is recorded on the host right before it finishes the synchronization routine and gives control back to the VM. Event d is generated as soon as the guest OS resumes execution. This model simulates property (1) as c indicates that the host is giving control to the VM and d represents its acknowledgement. Both these events hold $X + 1$ in their payloads to respect the one-to-one relationship.

The downside of this approach is the overhead introduced by the hypercall. Table 4.1 shows overhead measurements added by the hypercall, with and without tracing. However, registering to the system timer interrupt takes advantage of tickless kernels as they aim to reduce energy consumption by disabling interrupts on idle CPUs. In other words, the synchronization routine is not invoked on idle virtual machines, which otherwise would trigger a costly context switch on the host for no actual work.

Once traces are generated on both systems, the fully incremental convex hull algorithm is applied, which derives a synchronization function applied on all of the guest's timestamps. This approach is resistant to clock drifts as the convex-hull algorithm considers this issue and compensates for it in the generated formula. Additionally, it does not require `TSC_OFFSET` tracking or any other architecture-specific configuration.

Tableau 4.1 Overhead measurements

	Time (ns)		
	Without tracing	With tracing	Relative
One synchronization tracepoint	102	153	50.0%
Hypercall round-trip	5168	5565	7.7%

4.5.3 Synchronization results

To show the results of our trace synchronization algorithm, we traced simultaneously a running virtual machine and its host. We then merged the traces recorded from both systems and used TMF to view the result.

We show in Figure 4.3 the state of threads on different systems (the color legend is shown in Table 4.2). Thread `qemu:Debian` with TID 7030 serves as a virtual CPU of the VM as seen from the host. Events from the host's trace are used to recreate its state. Thread `wk-pulse` is a periodic CPU workload (in a pulse-like manner) running inside the VM. Therefore, events from the VM's trace are used to show its state. We can already expect that the vCPU of the virtual machine will follow a pulse-like pattern, as the guest system is mostly idle. On Figure 4.3, the staggered start of `wk-pulse` indicates a time gap of about 6 seconds between the host's and guest's clocks. We then used our synchronization algorithm to correct the guest trace's timestamps and reused the same view in TMF to view the result, as shown in Figure 4.4. We clearly see that `wk-pulse` is running on vCPU `qemu:Debian` because of their simultaneous state transitions. It is worth mentioning that the states of threads `qemu:Debian` and `wk-pulse` are computed independently from each other, yet they appear almost in perfect sync after applying the synchronization formula.

Tableau 4.2 Virtual Machine Analysis Color legend

Color	State
Green	Running
Yellow	Blocked
Orange	Preempted
Grayed out green	`PROCESS_VIRT_PREEMPT`

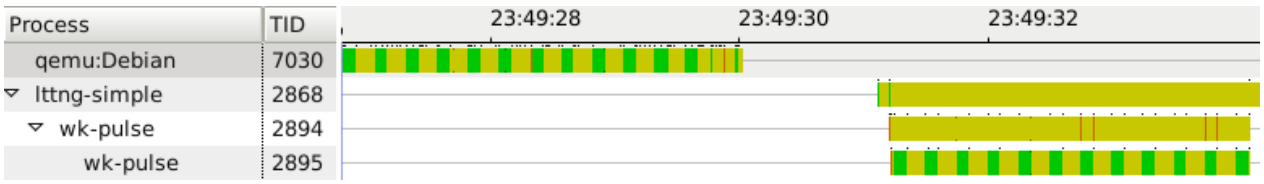

Figure 4.3 Merged traces without synchronization

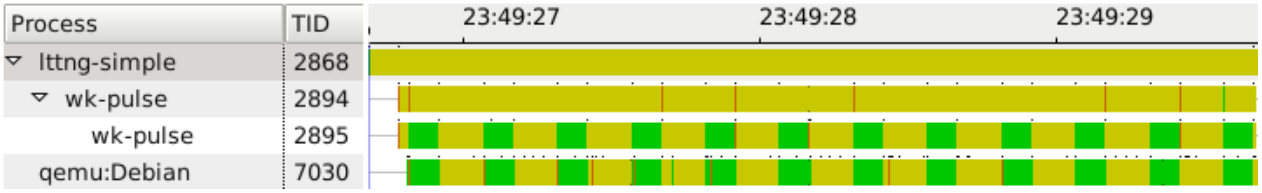

Figure 4.4 Merged traces with synchronization

4.6 Multi-Level Trace Analysis

This section presents how the state of each virtual CPU of a VM is recovered and rebuilt by analyzing the merged traces. The purpose of the analysis is to show the state of the vCPU throughout the trace as seen from the host. Our module parses the resulting trace and updates the attributes of the state system after each processed events. Moreover, we want to show the impact of preemption on the threads running within a VM. This analysis is useful as it shows the effective running time and execution of a thread compared to what is visible to the guest operating system. A vCPU at any time can be in one of the following states : `VMM`, `RUNNING`, `IDLE` or `PREEMPTED`. The state attribute of each vCPU can be found in the state system at path "`/Virtual Machines/VM Name/CPUs/vCPU ID/Status`. Figure 4.5 is a FSM (finite state machine) that shows transitions between these states. All of the events that trigger transitions originate from the host. Although not included in Figure 4.5, events from the virtual machines' traces are used to rebuild the states of the threads running on each vCPU within a VM. These threads can be found in the state system at paths "`/Virtual Machines/VM Name/Threads/TID/Status`". Following section 4.6.1 explains these states as well as the transitions by which they can be reached.

For clarity, we introduce the term pCPU which designates a physical CPU, as opposing to a vCPU which is in reality a QEMU thread emulating the CPU of a VM.

4.6.1 Virtual CPU States

VMM

State `VMM` represents the state when a QEMU thread is running hypervisor code instead of virtual machine code. In other words, it represents participation or involvement from the VMM, as to provide emulation, inject an interrupt into the guest's OS, or any other instruction requiring the external help of KVM. As explained in section 4.4.1, CPU transitions between non-root and root are instrumented with tracepoints `kvm_entry` and `kvm_exit` respectively. When a `kvm_exit` event is reached, the vCPU's state is set to `VMM` (transition 2).

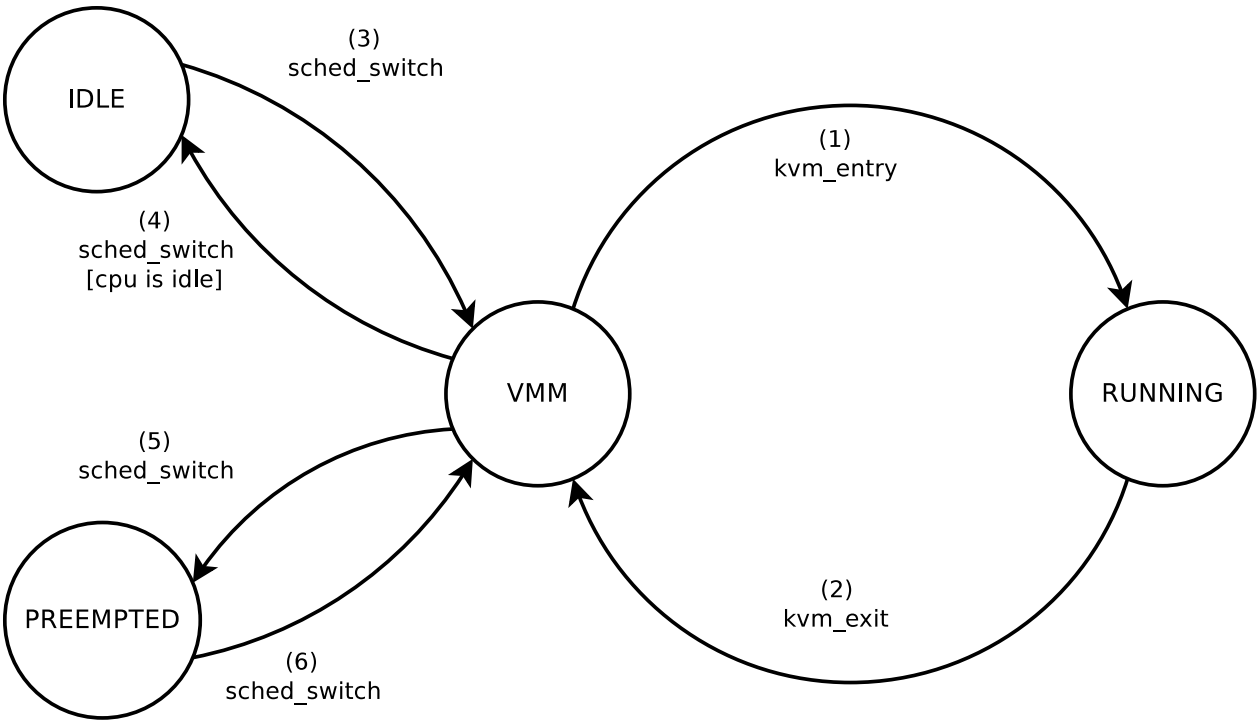

Figure 4.5 vCPU state transitions

On the other hand, it leaves this state on a `kvm_entry` event, returning to the state it was in prior to the VMM's involvement (transition 1). This state serves as an intermediate between any two states, as hypervisor cooperation is required for QEMU threads scheduling.

We also noticed that this state is reached everytime a QEMU thread is involved in a context switch, i.e., when a vCPU is scheduled out of a pCPU. Interestingly, when a QEMU thread is selected by the scheduler to run again, it first executes in the `VMM` state before explicitly invoking the `vmentry` instruction to give control to the guest's OS. This procedure is required because KVM needs to execute specific procedures regarding to the Virtual Machine Control Structure of the VM. KVM uses Linux's notifier chains to "register" on context switches involving a vCPU.
When a vCPU reaches this state, its current thread's status is set to `PROCESS_VIRT_PREEMPTED`, which designates wasted time due to the virtualization layer (we see it as preemption to execute hypervisor code, the thread is marked as "virtually preempted").

RUNNING

`RUNNING` shows execution of the VM's code. When in this state, a virtual CPU is considered as running without any involvement from the hypervisor, and instructions dedicated to a

specific vCPU are running directly on one of the host's pCPUs. For this state to be reached, two conditions must be satisfied. First, the QEMU thread emulating a vCPU must be in a running state on the host operating system. Secondly, in the guest operating system, the CPU associated to the specified QEMU thread must be in the running state as well, meaning that any process other than the idle task (`swapper`) is executing on the CPU.

IDLE

`IDLE` represents a state when a vCPU is not executing any code, and thus voluntarily yields the physical CPU. This state is reached when the QEMU thread emulating a vCPU is scheduled out of a pCPU, and if no thread other than the `idle` task is scheduled to run on this vCPU in the guest OS (transition 4). On Linux, the purpose of `swapper` (the `idle` task) is to invoke the scheduler to choose potential threads ready for execution, or to halt the CPU in case no thread is ready to run. The vCPU goes out of this state as soon as the thread emulating it gets scheduled back on the host (transition 3).

PREEMPTED

`PREEMPTED` is the state that indicates direct latency to the execution of a virtual machine. This state is reached when a vCPU is scheduled out of the pCPU by the host's scheduler (transition 5), while the vCPU was effectively serving a thread. Note that the running process on the vCPU stays in the `PROCESS_VIRT_PREEMPTED` state, which indicates that the vCPU on which the thread is running got preempted on the host operating system. Usually, this kind of information is not visible to a virtual machine, though it directly impacts the completion time of a task by introducing delays throughout the execution. As a result, a task may seem to complete in much longer than the effective time during which it was running. When scheduled back in (transition 6), the vCPU passes by the `VMM` state again to finally reach the `RUNNING` state and resume VM code execution.

4.6.2 Illustrative Example

We launched a thread that computes a Fibonacci sum on what appeared to be an idle virtual machine. The computer used was an Intel i7 (Nehalem) with 4 hyperthreaded cores (8 logical CPUs), 8 Gb of RAM, and running Debian Linux. Using `top`, no CPU-intensive thread was reported in the VM, and the *steal time* column showed a 0% vCPU preemption. Figure 4.6 shows the state of the Fibonacci task (thread `fibo`) as seen from the guest operating system. This view shows a monopoly of the CPU and a 100% utilization by the `fibo` task

for the whole duration of the trace. This state has been reconstructed by processing only the trace recorded on the guest.

Figure 4.6 View of Fibonacci experiment with traditional analysis

Figure 4.7 shows the result of our analysis module for the same experiment, after the host's and guest's traces merging and synchronization. We notice that vCPU 0 of the "Debian" VM is constantly transitioning between states `RUNNING` (green) and `PREEMPTED` (purple). With proper zooming, we can see `VMM` state as an intermediate for every transition. These transitions have direct repercussions on the execution of the `fibo` task, which is in turn moving between states `RUNNING` (green) and `PROCESS_VIRT_PREEMPTED` (grayed out green). With a quick look at the graphical view, we can see that the Fibonacci sum could potentially execute approximately twice as fast on a fully available pCPU, or less loaded host system.

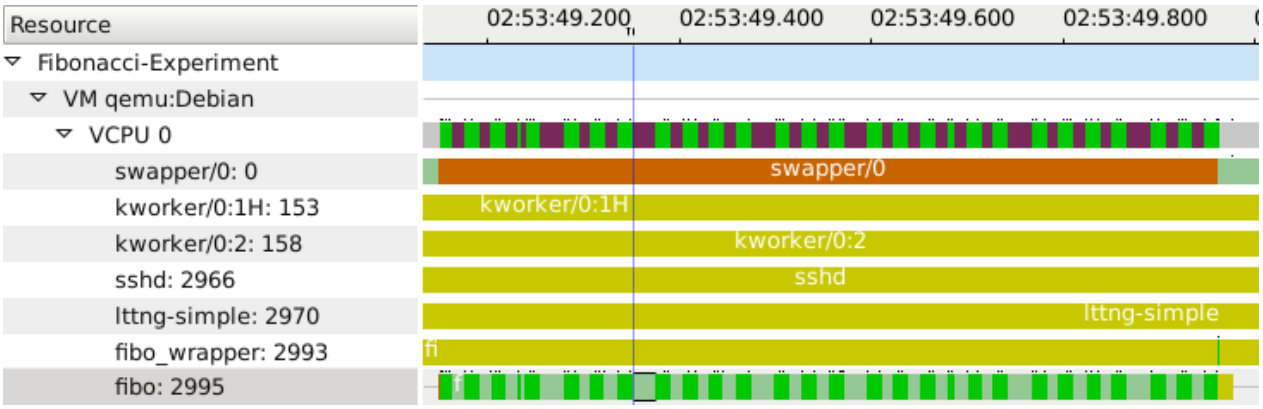

Figure 4.7 View of Fibonacci experiment with Virtual Machine analysis

The reason why `top` reported a 0% vCPU preemption (steal time), before starting the Fibonacci task, is because the vCPU was mostly idle. As a result, when it asks for CPU time, its request is immediately answered by the host's scheduler as it has the "highest priority" due to its idle nature. We can see that using such a tool to measure resource availability can actually be misleading. The only way to detect vCPU preemption using `top` would be to actively monitor the steal time while running the Fibonacci task.

4.6.3 Portability and flexibility

It is worth mentioning that the only KVM-specific tracepoints are `kvm_entry` and `kvm_exit`, which represent VMX mode transitions. Since these transitions are common to all hypervisors supporting hardware-assisted virtualization, our approach is therefore not specific to KVM. And, although these tracepoints are already included in Linux's source tree, they can be added in any hypervisor by simply instrumenting all calls to `vmentry` and `vmexit` instructions which requires very little effort, thus allowing this model to be used with any hypervisor. Moreover, if the administrator chooses not to instrument these transitions, little information would be lost, as the only state lost in Figure 4.5 would be `VMM`. Preemption and execution recovery would still be possible with little analysis precision lost (hypervisor involvement would account as effective CPU time instead of overhead due to virtualization). Furthermore, kernel traces generated from other operating systems can be used as well with minimal effort. As long as the events required to cover the FSM presented in section 4.6 are availbale, the model can be ported by simply specifying the names of these events. Moreover, in the case of microcomputers without hardware virtualization, the synchronization approach could potentially be extended to any other type of communication between the guest and the host, such as a TCP packet exchange. The rest of the analysis is based on the state system built, thus does not need reusing the traces.

4.7 Execution flow recovery

We now reach the second part of the analysis, which is to reconstruct the execution flow for a specific task of one of the virtual machines. The execution flow with regard to a certain task `A` is defined as the ordered set of execution intervals of all the tasks affecting the completion time of `A`. The purpose of the execution flow is to show detailed information about the execution of a certain thread as well as its interactions with other threads.

In the scenario shown in Figure 4.8, the execution flow is computed with regard to task `A`. The timeline shows the start and the end of the lifespan of this task, thus the analysis is time-bounded. In this example, it is clear that task `A` yields the CPU to allow execution of other tasks `B` and `C`. The scheduler then selects `A` after a certain amount of time, letting it complete its execution. Therefore, the completion of `A` was affected by the execution of `B` and `C`. When flattened, the execution intervals of all the threads to form one continuous execution interval which represents a busy CPU for the duration of the trace. Although this kind of information could be vaguely suspected from top-like tools, this level of detailed information is necessary for an advanced analysis of latency sources.

For a task executing inside a virtual machine, the computation of the execution flow

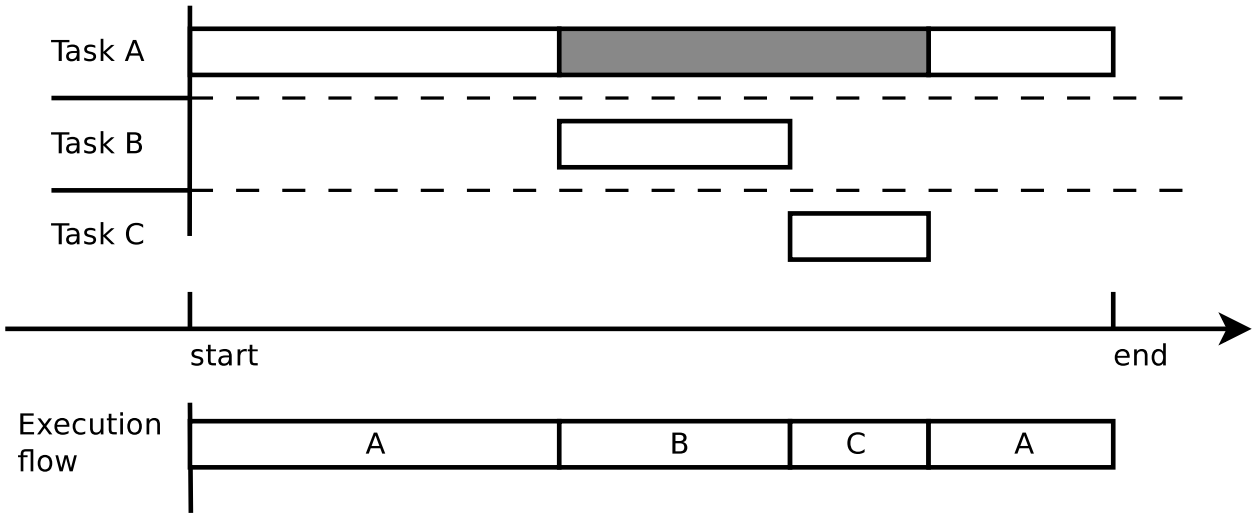

Figure 4.8 Simple example of an execution flow

should be adjusted to take into consideration interactions between different operating systems through the usage of shared resources. The objective of such an analysis is to provide detailed information not only about the execution of a certain task, but also about its interactions with other threads, whether they belong to the same VM, the host, or even a different VM. With such information, major causes of overhead can be easily tracked down by the host's administrator, and adjustments can be made to resolve the issues. Recovering the execution flow comes down to tracking all preemption events involving `A`. Causes of preemption can be within the same operating system and more obvious, or from different operating systems and hidden. In this section, we show that the execution flow recovery can be computed simply by querying the state system for key attributes modifications, without having to read the trace again.

4.7.1 Implementation

In this section, we call A the thread around which we want to recover the execution flow. The first step of the algorithm is to find all the entries involved in the execution flow according to task A. In figure 4.8, each of tasks A, B and C represent an entry. First, all the threads of all the systems are inserted as entries in the execution flow. This list of all the threads across systems can be recovered by parsing through attributes "`/Virtual Machines/*/Threads/*`" and "`/Host/Threads/*`" in the SHT. The second step of the algorithm is to compute the execution intervals of each entry with regard to task `A`. As a final step, we remove the entries that have minimal or no impact on the analyzed thread according to a minimal impact thre-

shold. The selection of the threshold doesn't affect the computing time of the algorihtm, as it is performed only after the whole algorithm has executed and all the durations have been computed. The impact of each thread can be measured using Equation 4.3.

To respect the relationship of affiliation between a thread and its system (host or VM), entries are stored in a tree-like structure with a depth of 2, where each node on the first level represents a system, and its children on the second level represent its threads.

As mentioned earlier, the execution flow can be represented with an ordered list of intervals, where each interval contains a start time, end time, a state, and the TID of the thread executing for the said interval. Algorithms 4 and 5 explain how this list can be built, recovering the execution flow with regard to task A.

Algorithm 4 is used to insert all intervals of A holding the "`RUNNING`" state. As a first step, we query the SHT to retrieve all modifications to the "`Status`" attribute of thread A. The SHT returns a list of intervals for different values of this attribute. Algorithm 4 parses this list and each interval holding the "`RUNNING`" state is directly inserted in the `result` list. However, for each interval holding the "`PREEMPTED`" value, a separate function is invoked to find which thread is preempting A. This function is shown in Algorithm 5.

Algorithm 4: Recovering the execution flow : inserting intervals in the `RUNNING` state

```
Input: StateHistoryTree s
List result;     // the list of execution intervals
StatusIntervals intervals = Query status intervals of A from s;

for each interval in intervals do
    currentPCpu = Query current pCPU of A;
    if interval.state == RUNNING then
        result.insert(interval);
    else
        result.insertAll(resolve(s, interval, currentPCpu));
    end
end

return result;
```

The `resolve` function requires an interval as well as a pCPU Id as input values. The work done by this routine is to find which thread is executing on the pCPU for the duration of the interval. In the case where the running thread is the vCPU of another VM, this function will then query the state system to get the running thread inside this VM. Once the running thread preempting A is deduced, it is returned to Algorithm 4 which will insert it in the

`result` list.

Algorithm 5: Function resolve() : Querying the state history tree to get the running threads while `A` is preempted

Input: *interval*
Input: *pCpu*
Output: *outList*

start = *interval*.start;
end = *interval*.end;
ThreadIntervals = Query "Current Thread" intervals of *pCpu* between *start* and *end*;
for *each interval t in ThreadIntervals* **do**
    if *t.tid is a vCPU* **then**
        intervals = Query "Current Thread" intervals of *t* between *t.start* and *t.end*
        outList.addAll(intervals);
    else
        outList.add(*t*);
    end
end
/* All intervals inserted in *outList* are in *RUNNING* state */
return *outList*;

Finally, in the `result` list returned by Algorithm 4, each interval "`interval`" of the list respects the following rules :

$$prevInterval.end = interval.start - 1$$
$$interval.end + 1 = nextInterval.start$$

Where `prevInterval` and `nextInterval` are respectively the previous and next intervals of `interval` in the ordered list `result` from Algorithm 4.

4.8 Use cases

This section shows how our work can be used in real-life cases to investigate latency in virtual machines. We start with a follow-up on the Fibonacci example introduced earlier (section 4.8.1). We then present different use cases that show either how to investigate a known issue (section 4.8.2), or a general analysis to verify normal execution of the system as a whole (4.8.3).

4.8.1 Follow-up on the Fibonacci Case

The first example we provide is the continuation of the Fibonacci use case presented earlier. Figure 4.7 showed that the Fibonacci took longer to execute due to preemption of the vCPU on which it executed. We follow-up on the matter by recovering the execution flow of the experiment. Figure 4.9 shows the result ; we can see that the preemption is due to a single CPU-intensive process on the host called `burnP6`.

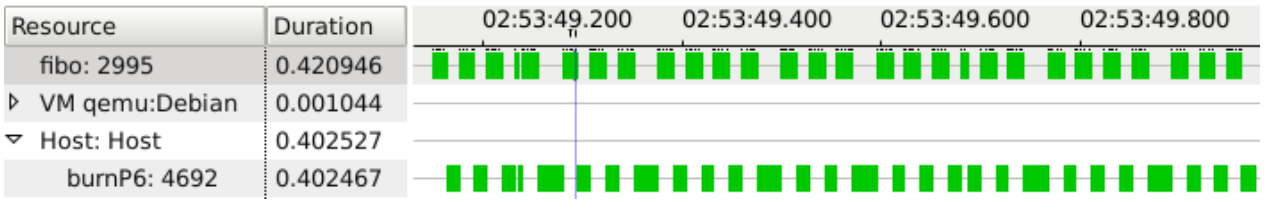

Figure 4.9 Execution flow recovery of previous Fibonacci experience

4.8.2 Investigation of Execution Anomalies

We now show a use case of a performance issue which can be easily tracked down using our graphical views. We developed a CPU-intensive task, named `critical_task` which computes for approximately 280 ms. A script spawns a `critical_task` thread in periodic fashion, every second, without waiting for completion. We traced the host and the guest operating systems simultaneously. Figure 4.10 shows the result of 6 `critical_task` threads. We can see the rate of thread forking at one thread per second. However, executions 3, 4, 5 and slightly 6 (threads 3523, 3525, 3527 and 3529) show abnormal computing duration although they appear as running (green) without interruption.

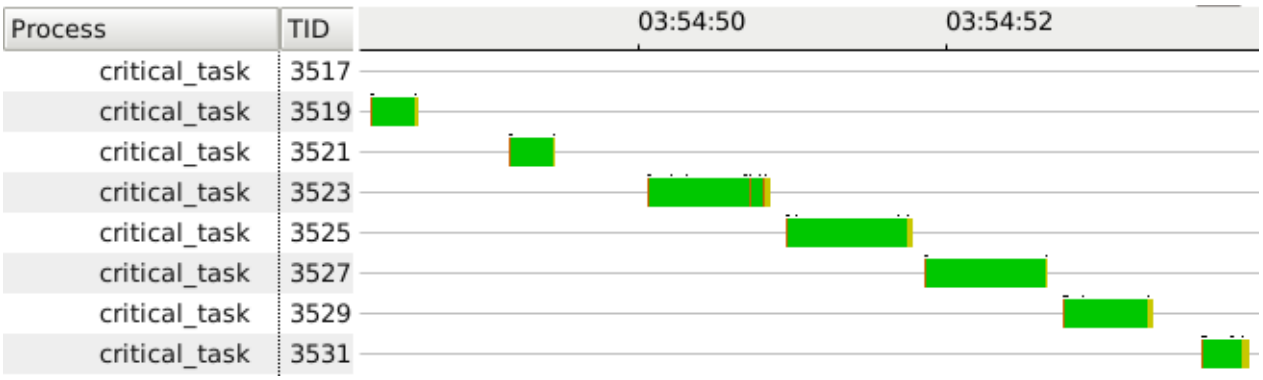

Figure 4.10 Execution of a periodic task as perceived by the VM
In some cases, execution inexplicably takes longer to compute
although the task appears as running

We then merged and synchronized kernel traces, and used our first graphical view to analyze the execution, as shown in Figure 4.11. On the vCPU 0 line, we see transitions between states `IDLE` (gray) and `RUNNING` (green) which indicates that the VM is mostly idle, except when `critical_task` threads are spawned. Additionally, we can clearly see latency on threads 3523, 3525, 3527 and 3529 (greyed out green) due to vCPU preemption (purple) for executions 3, 4, 5 and 6. This is an indicator that the VM is not allocated enough CPU time by the host's scheduler, and that the pCPU is strongly shared amongst host processes.

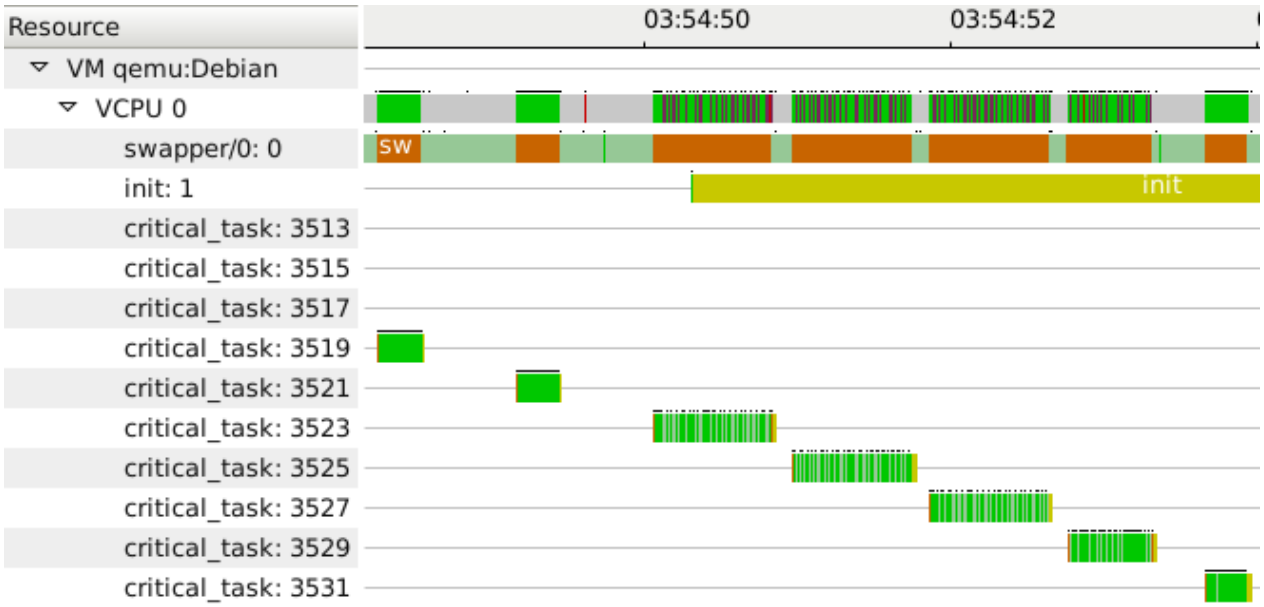

Figure 4.11 Using our view, we can see that the vCPU on which the critical task is running is actually being preempted on the host, which impacts the execution of the thread

Finally, we recovered the execution flow to investigate the source of this latency. The result is shown in Figure 4.12. The execution flow is centered around thread 3525, which seems to be the execution of the `critical_task` with most latency. The view shows that the pCPU is shared amongst three operating systems : Debian (the VM in which the critical tasks are running), Ubuntu (which is another virtual machine) and the host. We notice that threads `burnP6` from the host and `cc` from the Debian VM both strongly preempt the critical task, which explains its excessive duration. The "Duration" column shows the duration of the execution of each thread for the lifespan of thread 3525. For regular thread entries, this number represents the duration of execution of the particular thread. For system entries (non-leaf nodes), the "Duration" number indicates the sum of execution of all its threads. We can see that the critical task ran for 274 ms, the Ubuntu virtual machine ran for 270 ms and the host ran for 260 ms. These numbers indicate approximately a 33% usage of the CPU for each system, which indicates that the pCPU is strongly shared amongst them. Moreover, we see that process `irq/46-iwlwifi` executes for 296 us, indicating heavy network usage and packet processing. For each thread T, the proportion of time for which it preempted `A` is computed using Equation 4.3, where T_{out} is the timestamp indicating a scheduling out and T_{in} is the timsetamp indicating a scheduling in.

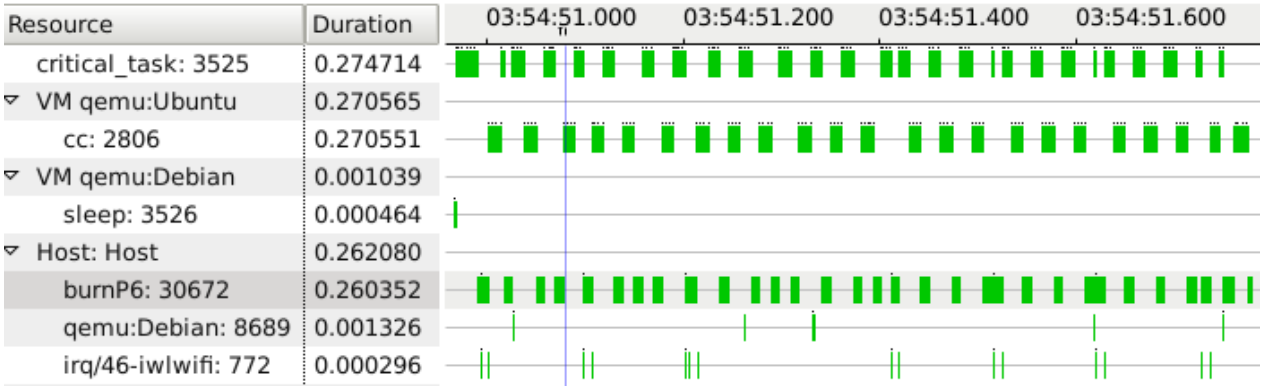

Figure 4.12 Execution flow recovery of problematic critical task

$$D(T) = \frac{\sum_{A.start}^{A.end} T_{out} - \sum_{A.start}^{A.end} T_{in}}{A.end - A.start} \tag{4.3}$$

4.8.3 Investigating a Residual Timer

We now present a use case that helped us investigate an unexpected operating systems problem. Figure 4.13 shows the result of our analysis for a workload similar to the one presented in the previous use case (section 4.8.2). We first see that the analyzed task is sharing the CPU with threads `cc` from the VM "Ubuntu" and `burnP6` from the host. However, for the second half of the analysis, the ciritcal task is being preempted by `swapper`, the idle task, from "Ubuntu". Such a behavior seems problematic as control is taken away from the analyzed thread to serve an idle thread. With a quick look at the trace when `swapper` is scheduled, we noticed events indicating the expiry of a timer. It turns out that a periodic timer was scheduled in the virtual machine, which would require CPU time for a very short period to acknowledge each timer expiration. This behavior introduces significant overhead as context switches are somewhat costly on the host system. To sum up, we saw how a "forgotten" timer in one virtual machine can affect the execution of other virtual machines. Such a problem can be easily fixed by the system administrator once it is located.

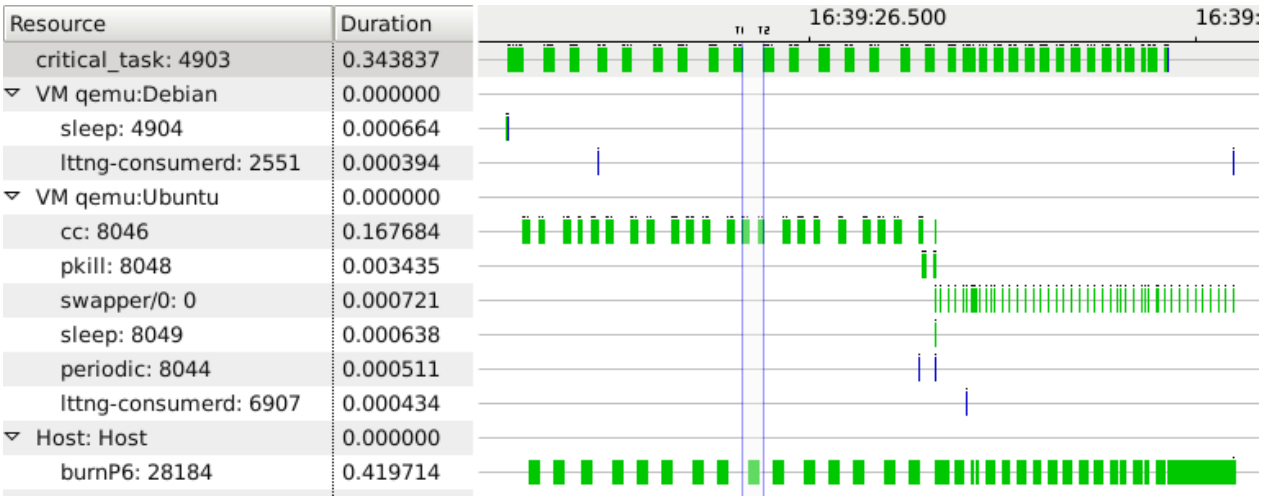

Figure 4.13 Execution flow recovery of problematic critical task with a periodic timer in another VM

4.9 Conclusion

Cloud computing and virtualization are evolving at a rapid pace. These emerging technologies created a need for analysis tools that can live up to the technological advance. In this chapter, we showed that kernel tracing can be used to analyze the execution of virtual machines under such conditions. We first proposed an approach to resolve the problem of clock drift and offset between operating systems. We then showed how the merged traces can be processed to rebuild the state of the virtual machines, as well as their vCPUs, throughout the trace. Finally, we explained how the execution flow with regard to a certain thread can be rebuilt for an in-depth analysis of its execution and interactions with other systems. All the solutions proposed in this paper were designed with requirements of portability and flexibility in mind. As a result, all the approaches explained are portable across operating systems, computer architectures, and complementary software (tracer and hypervisor).

4.10 Acknowledgements

The authors would like to thank Ericsson for their input to this study as well as for funding this research project. We also thank Geneviève Bastien for her help in the Open Source project TMF and Naser Ezzati Jivan for reviewing this paper.

CHAPITRE 5

DISCUSSION GÉNÉRALE

Nous présentons maintenant une discussion des résultats obtenus ainsi que du travail général réalisé au cours de ce projet.

5.1 Retour sur la synchronisation

Nous avons présenté précédemment l'approche que nous avons proposée pour remédier au problème d'incohérence des estampilles de temps entre des traces provenant de systèmes distincts. Les figures 3.2 et 3.3 montrent une tendance de divergence des horloges, d'où un besoin de recourir à un mécanisme de synchronisation au moment d'unir les traces. Dans le but de montrer le résultat de notre approche de synchronisation, nous avons généré à nouveau les mêmes courbes qui montrent la différence entre les horloges des différents systèmes pour des VM active et inactive en fonction du temps. Chacun des échantillons est obtenu avec la formule :

$$T_b - T_a \tag{5.1}$$

où T_a est le temps de l'évènement a et T_b est le temps de l'évènement b [1].

L'objectif de cette analyse est de montrer expérimentalement que l'algorithme de synchronisation résout le problème de divergence des horloges. Pour valider cette hypothèse, nous voulons montrer que la différence entre les horloges varie autour d'une valeur X (la moyenne) avec une marge d'erreur négligeable. Idéalement, cette valeur X devrait être le délai réel entre les évènements a et b. Les résultats sont présentés dans les figures 5.1 et 5.2 pour des machines virtuelles respectivement inactive et active.

1. les évènements a et b sont ceux utilisés pour la synchronisation et qui respectent la relation $a \rightarrow b$

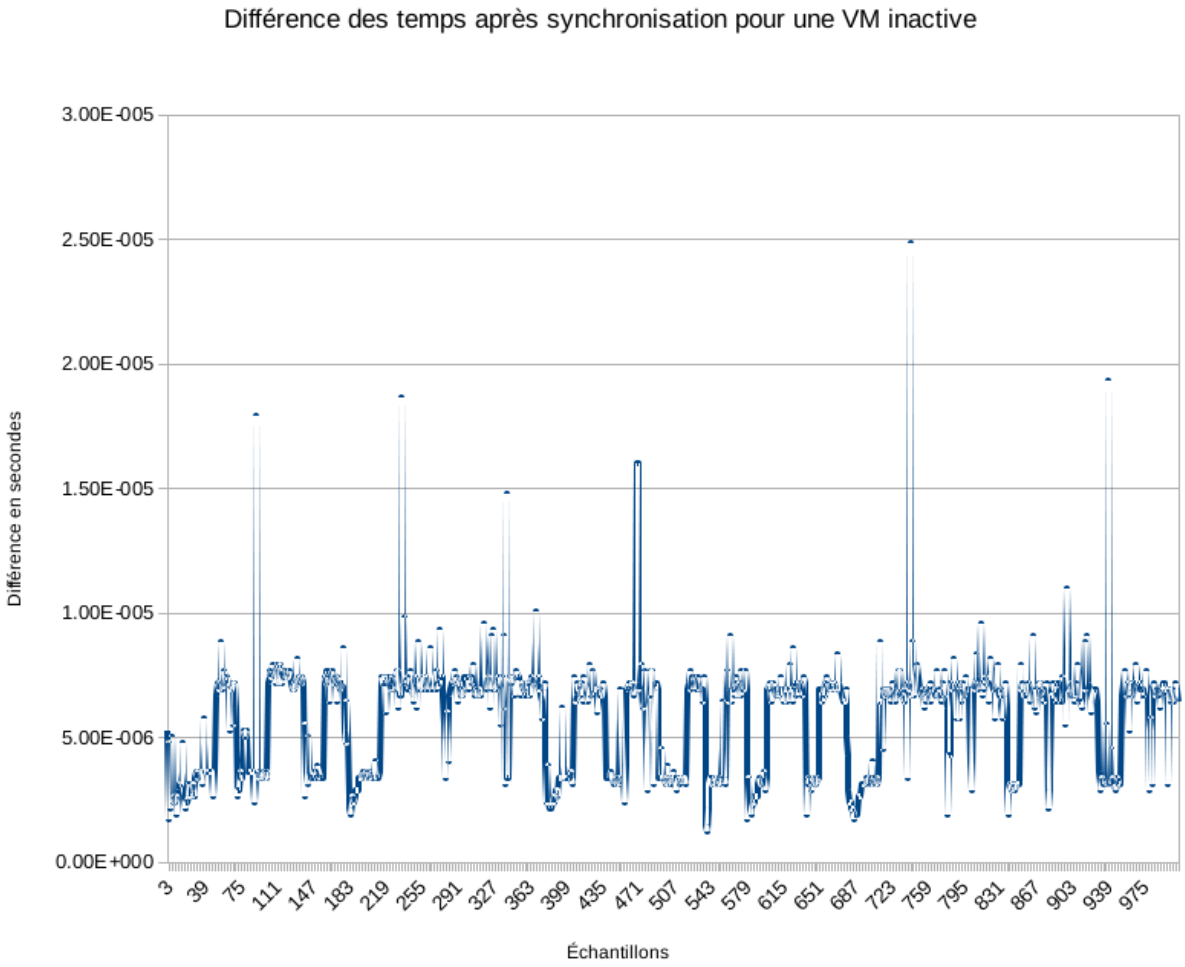

Figure 5.1 Différence entre les horloges des systèmes invité et hôte en fonction du temps après synchronisation pour une VM inactive

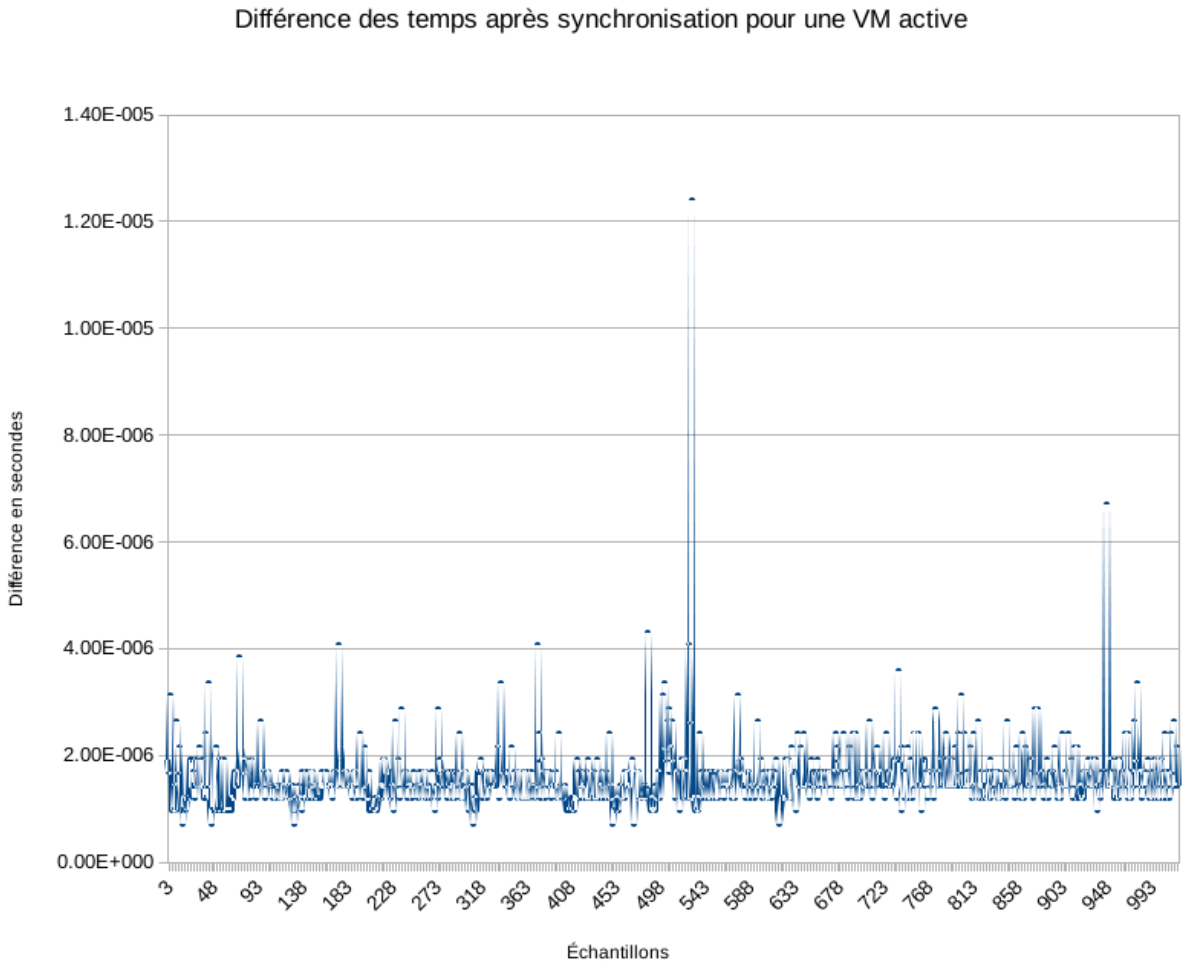

Figure 5.2 Différence entre les horloges des systèmes invité et hôte en fonction du temps après synchronisation pour une VM active

Deux remarques peuvent être tirées des figures 5.1 et 5.2. Tout d'abord, la tendance de divergence des horloges semble résolue après la synchronisation. En effet, bien qu'on note une variation dans la courbe, il semble exister une valeur moyenne autour de laquelle les points sont distribués. La prochaine section tente de confirmer cette hypothèse.

5.1.1 Analyse qualitative

Nous avons ensuite effectué une analyse qualitative de la population des différences des temps entre une VM et son système hôte. Nous avons généré une trace de 60 secondes dans une machine virtuelle active, ce qui génère environ 5000 échantillons. Tel que nous l'avons mentionné, nous tentons de montrer expérimentalement que la population varie autour d'une valeur X avec une faible marge d'erreur. Nous commençons par montrer un histogramme de la distribution des échantillons dans la figure 5.3.

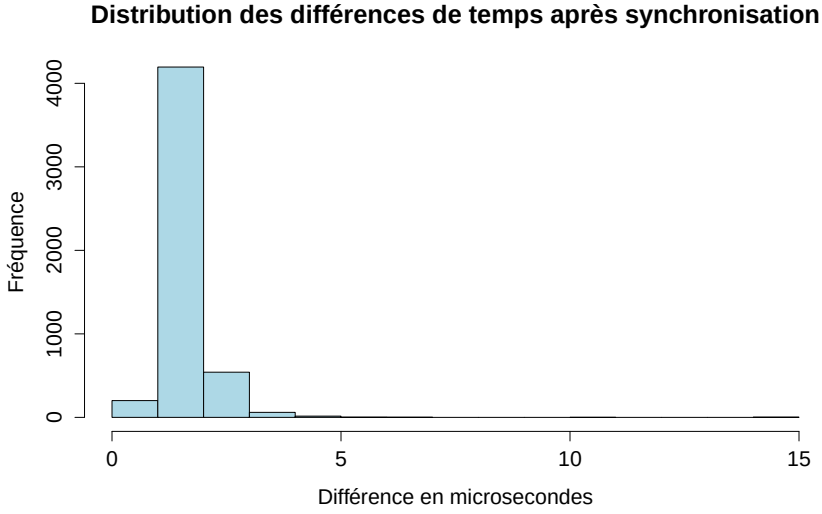

Figure 5.3 Distribution de la population

La figure 5.4 montre les résultats de l'analyse effectuée. Les points utilisés pour générer cette courbe sont obtenus en utilisant la formule

$$y = x - \overline{x} \tag{5.2}$$

où x est un échantillon et $\overline{x}$ est la moyenne. Le graphique est donc centré autour de 0 pour ne mettre en évidence que la marge d'erreur, qui est définie comme étant la distance entre x et $\overline{x}$. Notre objectif est de montrer que cette erreur est globalement non significative.

Dans ce graphique, 4481 échantillons (89.2%) se trouvent entre l'écart type et son op-

posé. De plus, 4903 (97.6%) échantillons sont bornés par le double de l'écart type et son opposé. Le tableau 5.1 montre des résultats additionnels. L'écart-type de 560 nanosecondes et l'intervalle de confiance calculé à 95% de 15 nanosecondes indiquent un fort regroupement de la population autour de la moyenne, tel que supposé précédemment. Ces valeurs sont un bon indice de la validité de l'algorithme de synchronisation qui arrive à corriger efficacement les estampilles de temps d'une machine virtuelle pour suivre l'horloge de l'hôte. En d'autres termes, la différence entre les horloges après la synchronisation fluctue légèrement autour de la moyenne calculée.

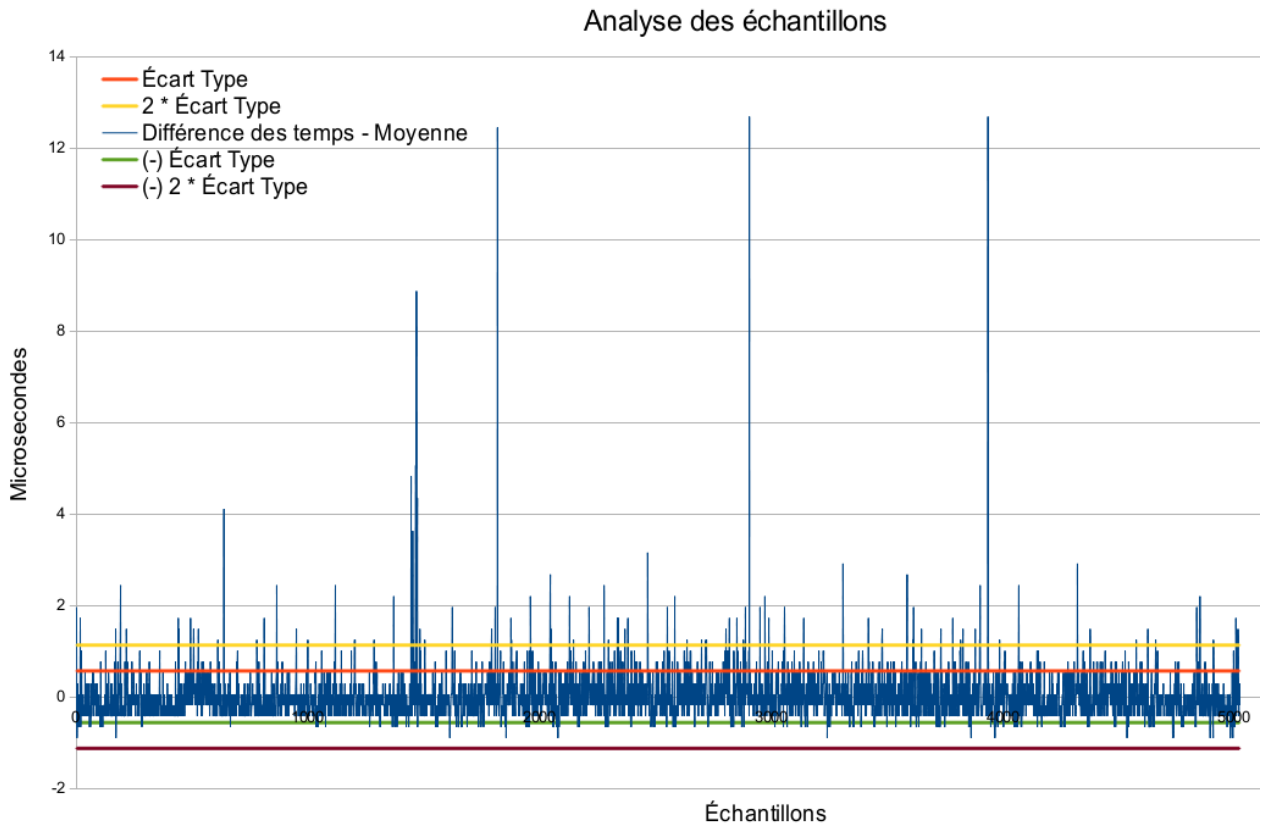

Figure 5.4 Analyse de la population de la différence des temps après synchronisation

Tableau 5.1 Analyse qualitative de la population

Métrique	Valeur
Taille de la population	5021
Moyenne (en microsecondes)	1.6163
Écart type	0.5639
Intervalle de confiance (95%)	0.0156

5.1.2 Analyse expérimentale

Lorsque l'hyperviseur interrompt la machine virtuelle pour un certain traitement, le système d'exploitation invité n'est plus en exécution. Il en est de même lorsque l'ordonnanceur de l'hôte arrête l'exécution d'un vCPU pour donner le CPU physique à un autre processus. Nous avons utilisé ces propriétés pour valider expérimentalement la synchronisation des traces. L'hypothèse à vérifier est donc qu'aucun évènement d'une trace d'un système invité ne peut survenir sur un vCPU lorsque celui-ci n'est pas ordonnancé sur un CPU physique, ou lorsqu'il est en mode *VMX root* (exécution de l'hyperviseur). Nous avons donc comparé le nombre d'évènements fautifs entre des traces avant et après synchronisation pour des VM actives et inactives. Un évènement fautif est un évènement qui apparaît comme étant enregistré dans une VM pendant que celle-ci ne s'exécute pas réellement sur le CPU physique. Pour calculer le nombre d'évènements fautifs, nous avons développé un algorithme qui s'exécute sur le résultat de l'union des traces. Les évènements de cette union sont traités l'un à la suite de l'autre. L'état de chaque vCPU (ordonnancé ou non) est sauvegardé en suivant les évènements `sched_switch` dont la source est la trace de l'hôte. Si la trace source de l'évènement traité provient d'un vCPU dont l'état est non ordonnacé, alors l'évènement est considéré comme étant fautif. Les résultats sont présentés sous forme de pourcentage dans le tableau 5.2.

Tableau 5.2 Analyse expérimentale de l'algorithme de synchronisation

	VM inactive	VM active
Avant synchronisation	65.5%	15.95%
Après synchronisation	0%	0%

Nous trouvons une valeur de 0% après synchronisation pour des machines virtuelles inactives et actives. En d'autres termes, aucun évènement fautif n'est trouvé après synchronisation, et tous les évènements des machines virtuelles se produisent légitimement, lorsque celle-ci est ordonnancée pour exécution sur l'hôte.

CHAPITRE 6

CONCLUSION ET RECOMMANDATIONS

Cette section se propose d'effectuer la synthèse des travaux présentés dans ce mémoire. Nous reviendrons sur les objectifs que nous nous sommes posés, et nous résumerons les résultats que nous avons obtenus. Nous discuterons ensuite des limitations de la solution proposée. Finalement, nous proposerons des améliorations potentielles à ce projet.

6.1 Synthèse des travaux

Au cours de ce projet, nous avons développé un modèle permettant d'analyser le comportement des applications dans des machines virtuelles alimentées par KVM. À partir de ce modèle, nous avons réussi à reconstruire le flot d'exécution centré sur un processus de la machine virtuelle pour étudier ses interactions avec les différentes machines virtuelles du même hôte. Tel que nous l'avons expliqué, TMF utilise un arbre d'attributs qui décrit l'état du système à tout moment de la trace. Nous avons établi des règles permettant de prendre en compte la couche de virtualisation lors de cette mise à jour, tel que présenté dans la figure 4.5. Cette partie du projet a donné naissance à deux vues graphiques dans TMF : une vue d'analyse de machines virtuelles ainsi que de leurs CPU virtuels et une vue de reconstruction du flot d'exécution.

Nous avons aussi proposé une méthode de synchronisation de traces à l'étape de leur union. Cette méthode est basée sur le jumelage d'évènements entre une paire de traces. Dans le but d'obtenir des évènements ayant une relation de causalité, nous avons ajouté des extensions au traceur sous la formes de modules qui doivent être chargés en mémoire par les systèmes d'exploitation actifs (systèmes hôte et invités). Ces modules utilisent des *hypercall*, qui sont des requêtes de collaboration entre les systèmes d'exploitation, pour obtenir une relation de causalité.

Finalement, nous avons réussi à reconstruire le flot d'exécution centré autour d'un processus d'une machine virtuelle. Le flot d'exécution montre les interactions entre les différents systèmes à travers le partage du CPU. Le flot d'exécution est complémentaire à l'analyse des VM, et montre la source des délais normalement invisibles depuis un système invité.

En résumé, ce projet a donné naissance à un modèle qui permet de transformer un modèle de bas niveau (une trace noyau) en une vue graphique de haut niveau. Cette vue graphique reconstruit l'état des machines virtuelles actives sur un poste physique. En utilisant ce modèle,

il est facile pour l'administrateur de l'hôte de repérer les baisses de performance des VM dues au surengagement du CPU. Lorsque plus de détails sont requis pour résoudre le problème, notre outil permet de reconstruire le flot d'exécution d'un certain processus pour obtenir les sources des délais introduits.

6.2 Limitations de la solution proposée

Nous discutons à présent des limitations des approches proposées pour les différents éléments de la problématique. Nous commençons par la méthode de synchronisation que nous avons implémentée sous la forme de modules additionnels à LTTng (*addons*). Outre le fait que ces modules ne sont pas intégrés à LTTng et doivent être maintenus séparément, un surcoût d'utilisation leur est attribué lorsqu'ils sont chargés par un noyau invité. En effet, nous avons expliqué que l'hyperviseur est invoqué par le biais *hypercall* à chaque sortie d'interruption logicielle (SoftIRQ). Ces transitions entre VM et VMM sont assez coûteuses, surtout lorsque les optimisations matérielles (TLB associatif, etc.) ne sont pas intégrées. Le plus grand inconvénient de cette approche est que les *hypercall* sont toujours exécutés par le système invité, même lorsque le traçage est désactivé. La seule méthode (courante) pour désactiver les invocations à l'hyperviseur est de décharger le module noyau.

Ensuite, les problèmes relatifs aux ressources physiques autres que le CPU ne sont pas facilement détectables avec notre modèle. Prenons comme exemple le cas du disque ; si plusieurs machines virtuelles accèdent à différents secteurs du disque simultanément, et que la *page cache* est en mode *guest only caching*, alors les délais dûs aux déplacements de la tête de lecture/écriture ne sont pas facilement détectables. Une façon d'investiguer ce genre de problème serait d'utiliser notre modèle pour détecter un blocage en attendant une ressource, puis analyser la trace en détail pour obtenir la raison du blocage.

6.3 Améliorations futures et projets connexes

Plusieurs extensions peuvent être apportées à ce projet. Tout d'abord, une étude expérimentale pourrait être entamée pour déterminer le taux minimal d'évènements de synchronisation requis pour permettre un résultat correct. Une fois ce taux obtenu, nous pourrions réduire le nombre d'*hypercall* sans affecter la précision de la synchronisation. Ceci aboutirait à une réduction du surcoût lié aux modules de synchronisation. Par ailleurs, il serait intéressant d'étendre LTTng pour tirer avantage de *virtio-trace*. Ce module de paravirtualisation destiné au partage de traces entre une VM et son hôte réduirait l'utilisation du disque dans la machine virtuelle lors du traçage. Selon la politique de gestion de la *page cache*, ceci pourrait être une optimisation générale de la machine virtuelle. De plus, cela éviterait l'utilisation du

réseau pour copier les traces à postériori du système invité vers le système hôte.

Aussi, une optimisation du module de synchronisation du système virtualisé pourrait être effectuée. L'enregistrement à l'évènement *softirq_exit* provoque un *hypercall* à chaque interruption logicielle, même lorsque le traçage est désactivé. Avec une meilleure intégration de ce module avec LTTng, nous pourrions enregistrer la routine de synchronisation uniquement lorsque le traçage est activé, et décharger le module lorsque le traçage est désactivé.

L'investigation de l'utilisation de la mémoire est un projet connexe intéressant. Tel qu'expliqué dans la section 2.5.6, le surengagement de la mémoire est un obstacle pour une virtualisation efficace de la mémoire. Lorsque le *ballooning* est utilisé pour remédier à ce problème, le choix de la VM dont la mémoire doit être réduite est un crucial. Une solution à base de traçage noyau pourrait être envisagée dans le but d'optimiser le partage de la mémoire du système entre les machines virtuelles. De plus, en instrumentant KSM avec des points de trace, des métriques intéressantes sur le partage de mémoire entre les VM pourraient être déduites.

Comme QEMU supporte maintenant le traçage avec LTTng, nous pourrions envisager de prendre avantage du traçage en mode utilisateur pour une analyse complémentaire à celle présentée dans ce projet. QEMU est responsable de l'émulation des périphériques et de l'interaction avec ceux-ci sur le système hôte. Le traçage de QEMU permettrait d'analyser l'utilisation des périphériques par une machine virtuelle. Par la suite, il serait possible de "caractériser" une machine virtuelle par son type d'utilisation, ou encore par le type de service qu'elle fournit. Par exemple, une machine virtuelle qui utilise fortement le disque pourrait supposément être un serveur de base de données. Une telle analyse serait fortement utile pour une distribution efficace des machines virtuelles dans un parc informatique en uniformisant le taux d'utilisation des ressources sur l'ensemble des nœuds physiques.

RÉFÉRENCES

[1] S. Rostedt, "Using the trace event macro." `http://lwn.net/Articles/379903`. Consulté en mars 2014.

[2] M. Desnoyers and M. R. Dagenais, "The lttng tracer : A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, pp. 209–224, Citeseer, 2006.

[3] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 65–81, 2012.

[4] M. Desnoyers, "Common trace format specification v1.8." `git://git.efficios.com/ctf.git`. Consulté en février 2014.

[5] J. Edge, "Perfcounters added to the mainline." `https://lwn.net/Articles/339361/`. Consulté en mars 2014.

[6] D. Toupin, "Using tracing to diagnose or monitor systems.," *IEEE software*, vol. 28, no. 1, 2011.

[7] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "State history tree : an incremental disk-based data structure for very large interval data," in *Social Computing (SocialCom), 2013 International Conference on*, pp. 716–724, IEEE, 2013.

[8] Z. Shao, L. He, Z. Lu, and H. Jin, "Vsa : An offline scheduling analyzer for xen virtual machine monitor," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2067–2076, 2013.

[9] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[10] N. Gandhewar and R. Sheikh, "Google android : An emerging software platform for mobile devices.," *International Journal on Computer Science & Engineering*, 2011.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[12] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*. Addison-Wesley, 2013.

[13] A. Kennedy and D. Syme, "Design and implementation of generics for the. net common language runtime," in *ACM SigPlan Notices*, vol. 36, pp. 1–12, ACM, 2001.

[14] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling java just in time," *Micro, IEEE*, vol. 17, no. 3, pp. 36–43, 1997.

[15] M. Helsley, "Lxc : Linux container tools." `http://www.ibm.com/developerworks/library/l-lxc-containers/`. Consulté en février 2014.

[16] P. Menage, "Cgroups documentation in the linux kernel." `Documentation/cgroups/cgroups.txt`. Noyau Linux version 3.13.

[17] B. des Ligneris, "Virtualization of linux based computers : the linux-vserver project," in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pp. 340–346, IEEE, 2005.

[18] A. Tucker and D. Comay, "Solaris zones : Operating system support for server consolidation.," in *Virtual Machine Research and Technology Symposium*, 2004.

[19] P.-H. Kamp and R. N. Watson, "Jails : Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, p. 116, 2000.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[21] A. Muller and S. Wilson, "Virtualization with vmware esx server," 2005.

[22] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm : the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[24] V. Oracle, "Virtualbox user manual," 2011.

[25] T. Naughton, G. Vallee, S. L. Scott, and F. Aderholdt, "Loadable hypervisor modules," in *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pp. 1–8, IEEE, 2010.

[26] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 2–13, 2006.

[27] V. Waldspurger, Carl, *Introduction to Virtual Machines*, 2010.

[28] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, ch. 23. No. 325462-045US, January 2013.

[29] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual Volume 2 : System Programming*, ch. 15. No. 24593, October 2013.

[30] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, ch. 24. No. 325462-045US, January 2013.

[31] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, 2007.

[32] A. M. Devices, *AMD-V Nested Paging*. July 2008. Revision 1.0.

[33] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The evolution of an x86 virtual machine monitor," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 3–18, 2010.

[34] C. Bae, J. R. Lange, and P. A. Dinda, "Comparing approaches to virtualized page translation in modern vmms," 2010.

[35] Advanced Micro Devices, "Amd-v nested paging," 2008.

[36] D. Bueso, E. Heymann, and M. A. Senar, "Towards efficient working set estimations in virtual machines," 2012.

[37] J. Corbet, "/dev/ksm : dynamic memory sharing." `http://lwn.net/Articles/306704`. Consulté en mars 2014.

[38] W. Stallings, *Operating systems Internals and design principles*, ch. 11. Prentice Hall, 7 ed., 2011.

[39] B. Singh, "Page/slab cache control in a virtualized environment," in *Proceedings of the Linux Symposium*, vol. 1, pp. 252–262, 2010.

[40] R. Russell, "virtio : towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[41] M. Tim Jones, "Virtio : An i/o virtualization framework for linux." `http://www.ibm.com/developerworks/library/l-virtio/`. Consulté en mars 2014.

[42] J. Corbet, "On vsyscalls and the vdso." `http://lwn.net/Articles/446528/`. Consulté en mars 2014.

[43] S. Seibold, "Add 32 bit vdso time function support." `https://lwn.net/Articles/529680/`. Consulté en mars 2014.

[44] Y. Yunomae, "Integrated trace using virtio-trace for a virtualization environment," in *LinuxCon North America/CloudOpen North America*, (New Orleans, LA), 2013.

[45] A. S. S. Masoume Jabbarifar, Michel Dagenais, "Online incremental clock synchronization," *Journal of the Network and Systems Management*, 2014.

[46] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, *et al.*, "Performance evaluation of virtualization technologies for server consolidation," *HP Labs Tec. Report*, 2007.

[47] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pp. 35–46, ACM, 2010.

[48] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[49] A. Anand, M. Dhingra, J. Lakshmi, and S. Nandy, "Resource usage monitoring for kvm based virtual machines," in *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, pp. 66–70, IEEE, 2012.

[50] A. Khandual, "Performance monitoring in linux kvm cloud environment," in *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, pp. 1–6, IEEE, 2012.

[51] J. Du, N. Sehrawat, and W. Zwaenepoel, "Performance profiling of virtual machines," in *ACM SIGPLAN Notices*, vol. 46, pp. 3–14, ACM, 2011.

[52] J. Fisher-Ogden, "Hardware support for efficient virtualization," *University of California, San Diego, Tech. Rep*, 2006.

[53] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "Efficient model to query and visualize the system states extracted from trace data," in *Runtime Verification*, pp. 219–234, Springer, 2013.